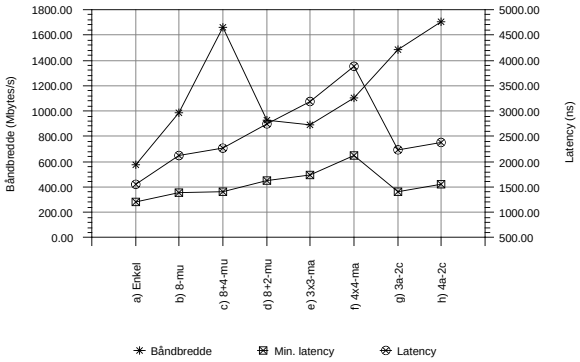
SCI-systemer med HIC som transportmedium



Hovedoppgave i informatikk
av
Thomas Waadeland

13. februar 1995

Institutt for informatikk
Universitetet i Oslo

Denne oppgaven er satt i 11pt Times ved hjelp av LaTeX.
Figurer er laget i Freehand 3.1 på Macintosh og idraw.
Grafer er laget med programmet wingz.

Forord

Dette er min hovedoppgave for cand. scient.-graden ved Institutt for Informatikk, Universitetet i Oslo.

Jeg vil takke mine veiledere Ernst Kristiansen, SINTEF og Fysisk institutt, og Øystein Gran Larsen, Institutt for informatikk, for all hjelp og nyttige diskusjoner.

Jeg vil også takke John Bothner, nå ved Dolphin, og Geir Horn, SINTEF, for deres bidrag. Ellers takk til de i newsgruppene uio.c og ifi.tex som har hjulpet til med henholdsvis C++ og LaTeX-koden, og takk til alle andre som har bidratt på forskjellig vis i form av råd og tips og ikke minst korrekturlesing.

Til slutt vil jeg også takke alle de nye vennene jeg har fått underveis. Studietiden hadde ikke vært det samme uten dere.

Thomas Waadeland

Innhold

Appendiks

Figurer

Tabeller

1

Introduksjon

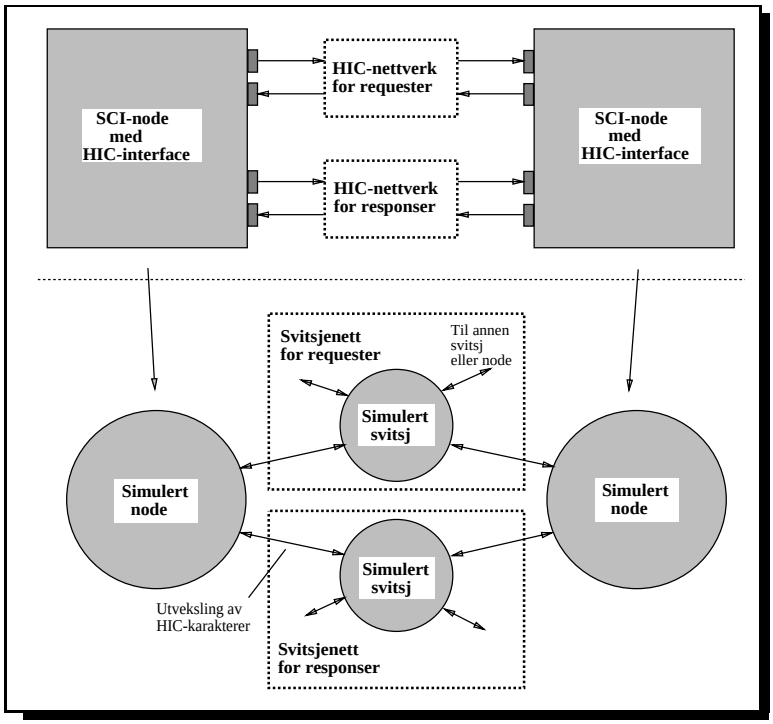
Flere av deltagerene ved utviklingen av SCI-standarden[1] har startet prosjekter med studier av forskjellige topologier for SCI-systemer. Det har vært aktiviteter rundt dette ved CERN i Genève, ved Universitetet i Wisconsin og i Oslo, hvor det har vært et samarbeid mellom Dolphin ICS, SINTEF Instrumentering og Institutt for informatikk og Fysisk institutt ved Universitetet. Tidligere simuleringer av SCI-systemer gjort i Oslo, har gitt tilnærmet like resultater som simuleringer av tilsvarende topologier andre steder [Bothner 94].

Høyhastighetsversjonen av SCI vil kunne være en essensiell byggesten for en rekke systemer. Disse SCI-systemene kan ha behov for å bli knyttet sammen. Derfor er det interessant å se på forskjellige måter å gjøre dette på. En mulig løsning kan være bruk av HIC[2], et serielt grensesnitt for utveksling av pakker.

Denne oppgaven tar for seg bruk av HIC som et transportmedium for SCI. Hovedmålene med oppgaven er å se på hvordan ytelsen i et slikt system blir med hensyn på effektiv båndbredde og latency, og hvilke nettverkstopologier som kan være gunstige. Videre er det interessant å se hvordan ytelsen blir sammenlignet med SCI-systemene undersøkt i [HulBot 93] og SCI-systemene med HIC-svitsj undersøkt i [Bothner 94].

For å se nærmere på dette, er det laget en simulator. En skisse av hvordan det virkelige systemet er modellert i simulatoren er gitt i figur 1.1. SCI-noden er modellert etter Dolphin's NodeChip[3]. HIC-interfacet er modellert etter BULLIT[4], som er en HIC-implementasjon fra BULL. HIC-svitsjen er modellert med utgangspunkt i ST C104[5].

[1] Scalable Coherent Interface [IEEE-SCI 92].
[2] Hetrogeneous InterConnect [IEEE-HIC 95].
[3] Dolphin GaAs NodeChip^{TM} [Dolphin 92].
[4] BULLIT [BULLIT 94].
[5] ST C104 asynchronous packet switch [ST C104 94].



Figur 1.1: Fra referent til simulert modell.

Denne oppgaven er delt inn i følgende kapitler:

Kapittel 2 gir en kort innføring i SCI for lesere som ikke er kjent med SCI.

Kapittel 3 gir en beskrivelse av HIC. Protokollen, pakkeformat, flytkontroll og definerte linker. Tilslutt litt om BULLIT.

Kapittel 4 tar for seg noen aspekter ved bruk av HIC som transportmedium for SCI, blant annet hvordan en SCI-pakke blir overført til HIC-format og problemer med deadlock.

Kapittel 5 beskriver HIC-svitsjen og hvilke rutingalgoritmer som er brukt. Siden det ikke fantes noen HIC-svitsj for HS-link da denne oppgaven ble påbegynnt, er det modellert en egen svitsj for bruk i simuleringene.

Kapittel 6 tar for seg selve simulatoren. Hvordan den er konstruert, dens forskjellige moduler og funksjoner, og hvordan den virker.

Kapittel 7 presenterer resultatene fra simuleringene, og gir en sammenligning med teoretiske verdier. Resultatene er også sett i forhold til hverandre og i forhold til andre arbeider gjort på området.

Kapittel 8 gir en kort konklusjon og oppsummering av hva som er gjort i oppgaven.

Appendiks A inneholder en ordliste.

Appendiks B forteller litt om programmeringsspråket C++, og hvorfor det er valgt.

Appendiks C forklarer bruk av simulatoren, HICsim. Opsjoner, konstanter og filsyntaks.

Appendiks D inneholder teoretiske utregninger for matrisenettverkene.

Ettersom jeg har valgt å skrive på norsk, oppstår det fort et problem med å finne norske ord i steden for engelske. Det er en del faguttrykk og navn på fenomener, som vanskelig lar seg oversette uten at det låter merkelig. Derfor har jeg valgt å bruke engelske navn og uttrykk der jeg synes dette faller naturlig. For eksempel har jeg brukt ord som «latency» og «deadlock». Enkelte steder der det er brukt norske oversettelser, er det engelske ordet satt i parentes. Noen ord og uttrykk er også forklart i appendiks A.

Tabeller og figurer hentet fra de forskjellige databladene, er stort sett beholdt i sin opprinnelige form.

I kapittel 6 hvor det er referert til navn på variabler, klasser og funksjoner i programmet, er navnet skrevet i *kursiv*.

2

Scalable Coherent Interface

IEEE standard 1596, Scalable Coherent Interface (SCI), ble utviklet på bakgrunn av erfaringene man hadde fått fra utviklingen av Fastbus[1], Futurebus+[2] og andre høyhastighets bus'er på slutten av 80-tallet. Man fant ut at bus'er ikke kunne oppfylle kravene for neste generasjon av multiprosessorer med hensyn på hastighet, skalerbarhet og muligheten for koherente hukommelses-systemer (memory systems). For å øke kommunikasjonsytelsen, måtte man derfor begynne å tenke i helt nye baner. Resultatet ble et pakkebasert kommunikasjonssystem over uavhengige punkt-til-punkt-linker.

I dette kapitlet er det gitt en kort beskrivelse av SCI. Først er det skrevet litt generelt om mål og hvordan kommunikasjonen fungerer. Protokollene og formatene er forklart, og til slutt er det gitt en beskrivelse av cache-koherens. Nærmere informasjon og mer detaljerte beskrivelser finnes i [IEEE-SCI 92]. I tillegg bygger dette kapitlet på [Dolphin 91] og [HulBot 93][3].

2.1 Generelt

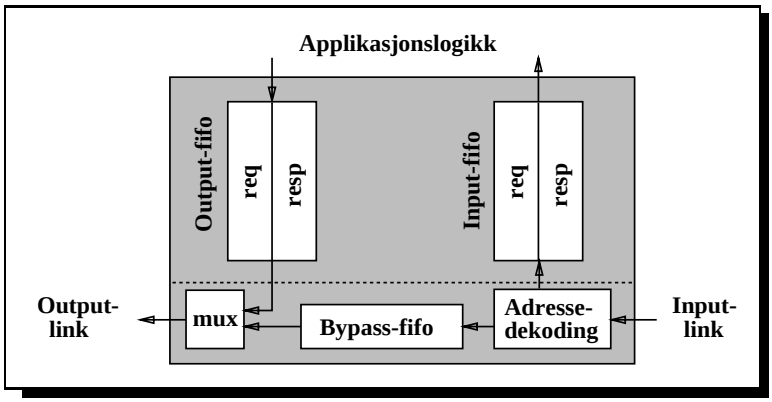
Målet med SCI var å definere et kommunikasjonssystem som kunne knytte sammen prosessorer og hukommelser i et multiprosessorsystem. Det skulle være mulig å bygge både små og store systemer til en rimelig pris (kost effektivt). For å få til dette, er protokollen laget enkel. Antall pinner på chip'en er lavt, men gir likevel muligheter for tilknytning til prosessorer, hukommelser, I/O og bus-adaptere til andre systemer. Høy ytelse og skalerbarhet er ivaretatt. Linkene har en hastighet på 1 Gbyte/s.

I et SCI-system foregår all kommunikasjon mellom nodene over punkt-til-punkt-linker. Hver node har en innkommende link (input-link) og en utgående link (output-link). Linkene består av 18 parallelle linjer. Av disse er 16 brukt til data og 2 til kontrolldata. Minste informasjonsenhet som blir overført på en SCI-link, er kalt et symbol, og består av 16 bit. Disse 16 bit'ene blir overført parallelt på linken. Noen av symbolene blir brukt til å danne pakker, andre blir brukt for å holde orden på kommunikasjonsprotokollen.

[1] ANSI/IEEE Standard 960-1993 + 1177-1993, ISBN 1-55937-396-2.
[2] ISO/IEC Standard 10857-1994 (IEEE 896.1 1994), ISBN 1-55937-373-3.
[3] Kapittel 5.

2.2 Protokollene

Kommunikasjonen mellom nodene er tilpasset med et sett av SCI-transaksjoner og protokoller, som inkluderer støtte for lesing og skriving av data, koherens, synkroniseringsmekanismer og meldingsutveksling (message passing). Alle transaksjoner blir sendt som SCI-pakker mellom avsender- og mottagernodene. Protokollene er nødvendige for å håndtere flytkontroll og feilretting (error recovery) og for å unngå deadlock.



Figur 2.1: SCI-node med fifo'er.

En SCI-node skal kunne motta data på innkommende link samtidig som den sender data på utgående link. Dette er løst ved bruk av fifo'er (figur 2.1). Disse sørger også for kompensasjon mellom den høye hastigheten på linken og den lavere hastigheten i noden. Bypass-fifo blir brukt til å lagre pakker adressert til andre noder, mens noden sender fra sin egen output-fifo. Input-fifo tar i mot pakker adressert til denne noden. For detaljer, se [IEEE-SCI 92].

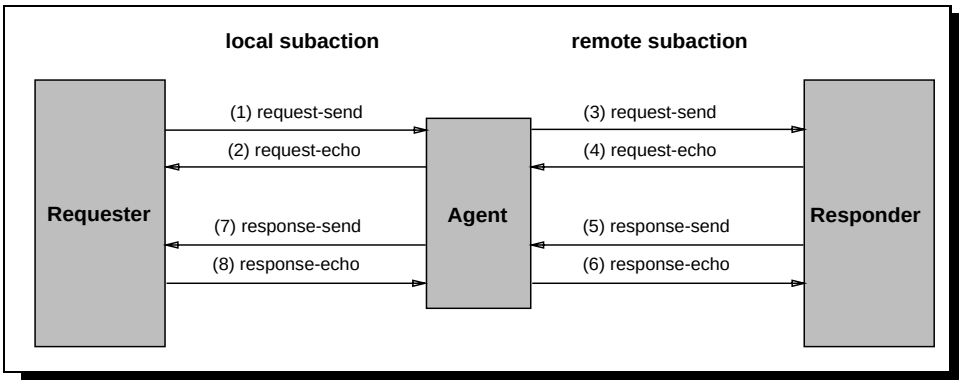
2.2.1 Transaksjonene

All kommunikasjon i SCI er definert som transaksjoner. En transaksjon starter med en «request» og slutter med en «response». Hver node i systemet kan ha response og/eller request muligheter.

Når en transaksjon foregår mellom to SCI-ringer, er det kalt en fjerntransaksjon. Dette involverer bruk av en agent, som kan være en svitsj mellom to SCI-ringer, eller en bro eller et interface til et annet system, for eksempel HIC som det er sett på i denne oppgaven. Når agenten tar ut en pakke fra en ring, tar den ansvaret for å sende den videre og gir et lokalt «echo» til avsenderen. Figur 2.2 viser et eksempel på fjerntransaksjon. For nærmere beskrivelse av transaksjonsformatene henvises det til [IEEE-SCI 92].

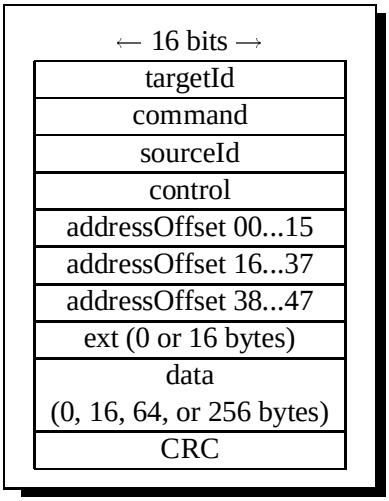
2.2.2 Pakkeformatene

SCI har mange pakketyper. Hovedtypene er request-, response- og echo-pakker. Pakkeformatene er vist nedenfor. For simuleringene i denne oppgaven er det kun



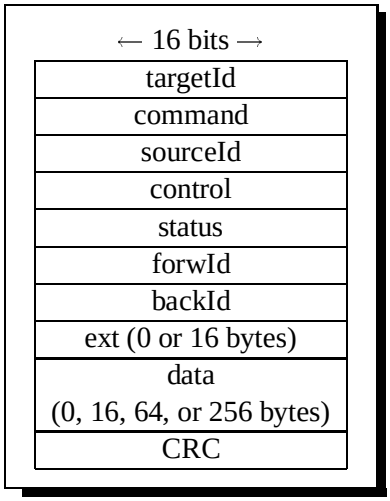
Figur 2.2: Eksempel på en fjerntransaksjon. Transaksjonene utføres i rekkefølge fra (1) til (8). Agenten sørger for videre levering av pakka og gir derfor et lokalt echo. Hentet fra [IEEE-SCI 92].

mottager- og avsenderadressene som er interessante. Datafeltet er i simulatoren satt til 64 bytes.



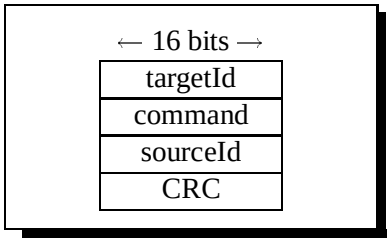
Figur 2.3: Request-Send pakkeformat.

Request-Send pakkeformat (figur 2.3)
Beskrivelse av de forskjellige feltene i pakka:
targetId – er adressen til noden pakka blir sendt til, mottageradressen.
command – angir pakketypen og gir flytkontroll informasjon.
sourceId – er adressen til avsendernoden.
control – gir informasjon om global systemtid, prioritet og transaksjonsid.
addressOffset – er internadresser i avsendernoden.
ext – er for «extended header».
data – inneholder 0, 16, 64 eller 256 bytes data.
CRC – blir brukt til feildeteksjon, (Cyclic Redundancy Check).



Figur 2.4: Response-Send pakkeformat.

Response-Send pakkeformat (figur 2.4)
Feltene i response-pakka er stort sett lik feltene i request-pakka, med unntak av feltene:
status – gir transaksjons-status, inklusive cache-koherensinformasjon.
forwId og *backId* – blir brukt av cache-koherensprotokollen.



Figur 2.5: Echo pakkeformat.

Echo pakkeformat (figur 2.5)
Feltene i echo-pakka tilsvarer stort sett feltene i request-pakka.

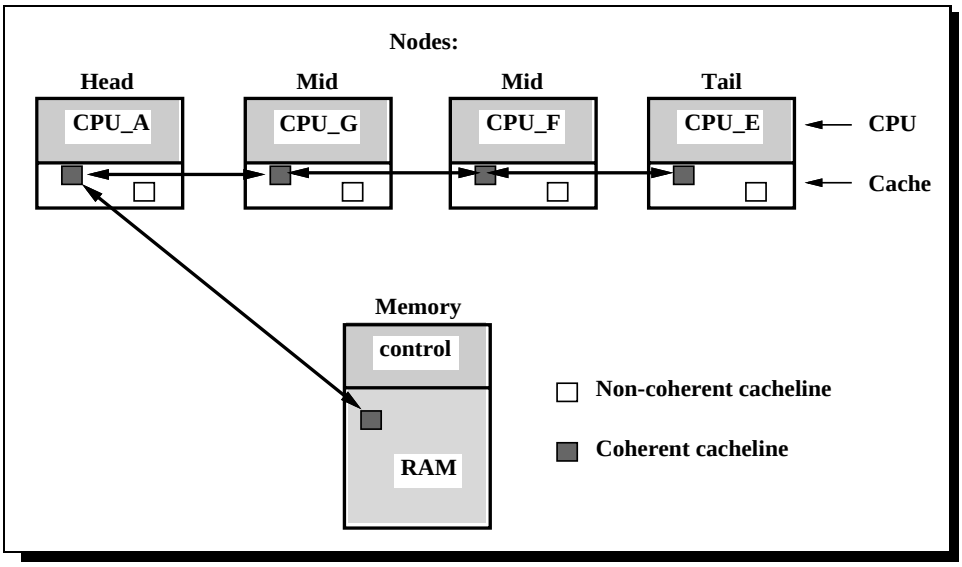
2.3 Cache-koherens

Nesten alle dagens prosessorer bruker cache for å redusere tidsforbruket ved hukommelsesaksessering. En cache er en liten, lokal høyhastighetshukommelse, hvor prosessoren lagrer data midlertidig. Når flere prosessorer har hentet samme del av en felles hukommelse til sin cache, skaper det problemer hvis en prosessor forandrer på noe i sin lokale cache. Dette er kalt «cache-koherens problemet» og er i SCI

løst ved hjelp av distribuerte lister (distributed directories), også kalt lenkede lister (chained directories).

En slik distribuert liste passer godt sammen med ønsket om skalerbarhet med hensyn på antall noder. I teorien er det ingen øvre grense for antall noder i lista, men tidsforbruket vil øke når lista blir lang. All cache-informasjon går bare til nodene som er involvert i lista. Oppdatering av lista blir delt mellom nodene og er ikke overlatt til en felles hukommelsestyringsenhet (memory management unit).

Lista er organisert som en dobbel-linket pekerkjede. Den inneholder alle nodene som deler samme del av en hukommelse. Operasjoner på lista blir enklere ved å ha en dobbel-linket liste. Et eksempel på en slik liste er vist i figur 2.6. Plasseringen av nodene i figuren har ingen sammenheng med hvordan nodene er knyttet sammen rent fysisk. Hver node i lista kan lese data fra cache'en, men det er bare den første noden i lista som har skriveaksess. Hvis en ny node ønsker å skrive, blir lista fjernet. De andre nodene oppdaterer dataene som er lagret i sine cache'er, ved å danne en ny liste. For en nærmere beskrivelse se [IEEE-SCI 92].



Figur 2.6: Eksempel på distribuert cache-liste. Alle nodene som har en kopi av samme del av en hukommelse (memory) i sin cache, er lenket sammen i en liste. Hentet fra [IEEE-SCI 92].

3

Hetrogeneous InterConnect

Hetrogeneous InterConnect (HIC) er beskrevet i IEEE's utkast til standard P1355, [IEEE-HIC 95], og er et resultat av ESPRIT-prosjektet OMI/HIC (Open Microprocessor systems Initiative – high performance Hetrogeneous Interprocessor Communication).

I dette kapitlet er det gitt en enkel innføring i HIC, slik at leseren skal kunne forstå hva som foregår i simulatoren. Det er skrevet litt om målene og hensikten bak HIC, selve HIC-protokollen, og de forskjellige linkene standarden definerer. Det er også gitt en kort beskrivelse av BULLIT, som er BULL's implementasjon av HIC's HS-link. BULLIT er i denne oppgaven brukt til å modellere HIC-interfacet til SCI-nodene. For detaljer henvises det til [IEEE-HIC 95] og [BULLIT 94].

3.1 Generelt

Eksisterende standarer på området var laget for lange distanser eller for å oppnå ekstreme ytelser. Noe som ga høye kostnader. Eller det var standarder for bus'er som ga både begrenset ytelse og skalerbarhet.

Målet med HIC-standarden er derfor å lage en standard for raske, parallellprosesserings-systemer med lave kostnader og høy ytelse. HIC må være rimelig for at det ikke skal være en dominerende utgift i det totale systemet. I tillegg ønsker man lav latency for at HIC ikke skal være noen begrensende faktor. For at ytelseskostnadene skal stå i forhold til systemets størrelse, må HIC også være skalerbart.

HIC sørger for et transparent transportlag av høynivåprotokoller for kommunikasjon, meldingsutveksling eller delte hukommelsestransaksjoner og gir støtte for linker mellom heterogene systemer.

HIC-standarden definerer fysiske koblinger og kabler, elektrisk utstyr og logiske protokoller for serielle og skalerbare punkt-til-punkt-forbindelser. HIC definerer forbindelser i kobber og fiberoptisk teknologi for hastighetene 10-200 Mbaud, 1 og 3 Gbaud.

3.2 Protokollen

Standarden definerer en pakkeprotokoll og hvordan denne blir implementert på forskjellige fysiske medier. Dette omfatter både et fysisk (avsnitt 3.3) og et logisk lag.

Det logiske laget er definert som en protokollstack. Denne er delt i lagene SL (Signal Level), CL (Character Level), EL (Exchange Level), PL (Packet Level) og TL (Transaction Level). Standarden definerer representasjoner for hvert lag som er spesifisert i forhold til det underliggende laget, sammen med regler for utvekslingen av disse representasjonene mellom hvert lag.

3.2.1 SL-laget

SL-laget tar seg av selve signalkodingen. Et signal er definert som en fysisk målbar kvantitet som varierer i tid, for å overføre informasjon. Signalet er ordnet som en bit-sekvens.

3.2.2 CL-laget

I CL-laget blir flere bit gruppert og sendt sammen i en såkalt karakter (character). En karakter er kodet slik at den sikrer DC-balansen og påliteligheten på linken. Selve kodingen avhenger av typen fysisk link. Linkene blir senere beskrevet i avsnitt 3.3.

3.2.3 EL-laget

EL-laget beskriver prosedyrene for utveksling av karakterer mellom nodene. Det er definert to typer karakterer, normal-karakter (n-karakter/n-char) og «level»-karakter (l-karakter/l-char). l-karakterene er kontroll-karakterer som blir brukt til flytkontroll, oppstart, nedtak og feildeteksjon. n-karakterer er data-karakterer og «end-of-packet»-karakter (EOP).

Simulatoren utviklet i denne oppgaven, opererer på dette laget. De simulerte nodene utveksler n- og l-karakterer. En skisse av EL-lagets kommunikasjon mellom to noder er gitt i figur 3.1.

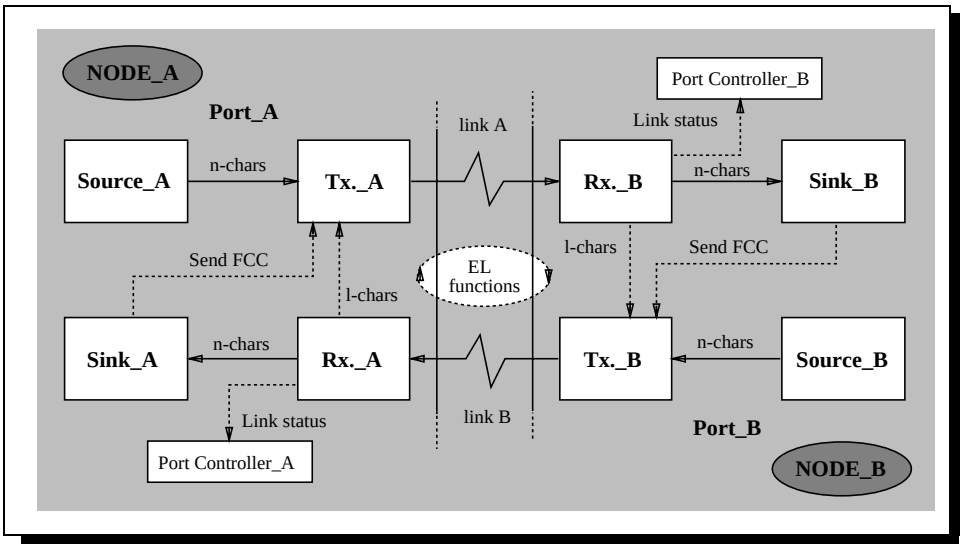
Flytkontroll

En viktig del av EL-laget er å ta seg av flytkontrollen. Flytkontrollen sørger for at fifo'ene ikke blir fulle og at ingen n-karakteter forsvinner.

Avsender har en viss kreditt som indikerer hvor mange blokker med n-karakterer den får sende uten å overfylle mottagerens fifo. En blokk består av F antall n-karakterer. For hver n-karakter som blir sendt, blir en teller økt. Når telleren når F, minker kreditten med 1, og telleren blir satt til 0. Når avsenders kreditt er 0, sender den kun l-karakterer. Avsender må da vente til den mottar en «Flow control link character» (FCC) fra mottager. For hver FCC øker kreditten med 1 igjen. Mottager sender en FCC med en gang den har plass til å motta en blokk på F antall n-karakterer. En implementasjon av FCC-funksjonene blir senere vist i avsnitt 3.4, figur 3.6.

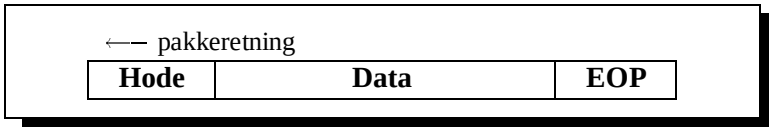
3.2.4 PL-laget

I PL-laget blir n-karakterene ordnet i pakker. En pakke er en sekvens av n-karakterer med spesifikk rekkefølge og format. Ordnede karakterer fra forskjellige pakker kan ikke bli blandet på en link. En pakke består av et hode (header) med mottageradresse fulgt av data (payload) og slutter med et end-of-packet-merke (EOP).



Figur 3.1: EL-lagets kommunikasjon mellom to noder. Nodene har en avsender (Tx.) og en mottager (Rx.) som holder orden på sending og mottak av n- og l-karakterer. Hentet fra [IEEE-HIC 95].

HIC-protokollen forutsetter bruken av et pakkesvitsjet nettverk, hvor nødvendig rutinginformasjon for å få pakken korrekt gjennom nettet ligger i de første n-karakterene (hodet) av pakka. Selve rutingen foregår i PL-laget. Det er ingen maksimumslengde på pakkene. Pakker som kommer etter hverandre på en link, kan ha forskjellige mottagere. Hver pakke blir overført i sin helhet. Det vil si at overføringen av en pakke på en link må være avsluttet før neste pakke kan bli sendt. Pakker kan, avhengig av nettverket, ankomme mottageren uten å komme i rekkefølge. Pakkeformatet er vist i figur 3.2.



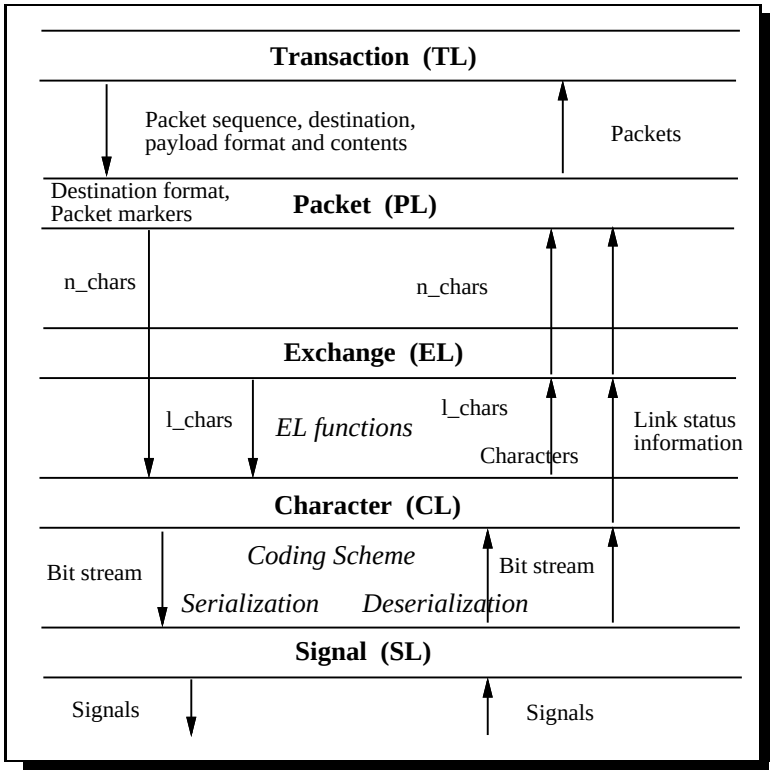
Figur 3.2: HIC's pakkeformat.

- *Hode* består av én eller flere karakterer med mottageradresser som blir brukt til rutingen, slik at nettet kan overføre pakka fra avsender til riktig mottager. HIC-standarden definerer ikke hva slags rutingalgoritme som blir brukt.
- *Data* er selve dataene i pakka. Her kan brukeren selv bestemme hva som skal bli overført. For eksempel som i denne oppgaven hvor datafeltet er en SCI-pakke (kapittel 4).
- *EOP* markerer slutten på pakka (end-of-packet). Første n-karakter som blir mottatt etter en EOP, blir mottatt som hodet i en ny pakke.

3.2.5 TL-laget

En transaksjon er spesifisert som en sekvens av pakker mellom to eller flere noder for å utføre noe. Hovedfunksjonen til TL-laget er å spesifisere dataformatet, og hvordan utvekslingen av pakker skal foregå. Man kan si at PL- og TL-laget er grensesnittet mot brukeren. I denne oppgaven vil TL-laget typisk være definert av SCI-protokollen (kapittel 2).

En hierarkisk fremstilling av de forskjellige lagene og hvordan de fungerer sammen er vist i figur 3.3 .



Figur 3.3: HIC's protokollstack. Hentet fra [IEEE-HIC 95].

3.3 Definerte linker

Det fysiske laget definerer implementasjonen av protokollen på fysiske medier. HIC-standarden spesifiserer tre forskjellige serielle linkforbindelser: DS- (Data/Strobe), TS- («3-to-6») og HS-link (High Speed). Noen spesifikasjoner ved HS-link er gitt i avsnitt 3.3.1. Se ellers [IEEE-HIC 95] for detaljer. De forskjellige linkimplementasjonene for HIC varierer i teknologi, hastighet, transmisjonsmedium og signal konvensjon. Linkene er listet opp i tabell 3.1.

Type	Transm.-medium	Baud rate	Data båndbr.	Transm.-type	Maks avst.
DS-SE-02	Single-ended el.	1M–200M Baud	35 MBytes/s	4 wires	1 m
DS-DE-02	Differential electr.	1M–200M Baud	35 MBytes/s	8 wires	30 m
TS-FO-02	Optic fibre	250M Baud	35 MBytes/s	2 multi-mode fibres	300 m
HS-SE-10	Single-ended el.	700M–1G Baud	160 MBytes/s	2 coax cables	5 m
HS-FO-10	Optic fibre	700M–1G Baud	160 MBytes/s	2 multi-mode fibres	100 m
				2 mono-mode fibres	500 m

Tabell 3.1: HIC's definerte linker.

3.3.1 HS-link

HS-linken er ment brukt til høyhastighetsforbindelser, som kan ha hastigheter opp til 1 Gbit/s. Den fiberoptiske HS-linken skiller seg bare fra coax-versjonen på det fysiske laget.

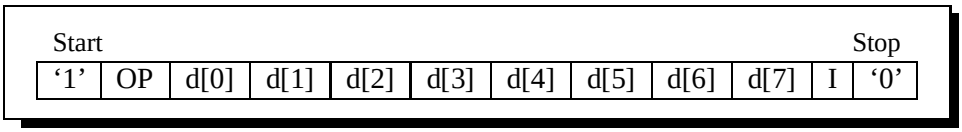
HS-linken er asynkron, det vil si at klokkeinformasjonen er kodet inn i selve signalet, data strømmen, slik at mottageren kan justere sin egen klokke ut fra klokkinga i signalet den mottar. For å få til dette, starter hver byte med en logisk '1' (Start bit) og slutter med en logisk '0' (Stop bit). I tillegg til disse bit'ene er det lagt på et paritetsbit (Odd Parity) og et inverteringsbit (Invert). Dette gir såkalt 8B/12B DC-koding, det vil si at 8 databit er kodet og sendt som 12 bit.

8B/12B-koding

Som nevnt, blir én databyte utvidet med fire bit, to bit er lagt til for å justere klokka, ett for paritet (OP) og ett inverteringsbit (I) for å ivareta DC-balansen. I figur 3.4 er det vist hvordan en HIC-karakter blir bygd opp og sendt på en HS-link. Bit'ene blir sendt i rekkefølge fra Start til Stop. d[0]-d[7] er databit'ene, hvor d[7] er det mest signifikante bit'et. Paritetsbit'et er satt inn for å kunne detektere enkle bitfeil som oppstår under transmisjonen.

Kontroll-karakterer (l-karakterer)

8B/12B-kodingen gir 256 data- og 126 kontroll-karakterer. l-karakterene blir skilt fra vanlig data (n-karakterer) med et kontroll-/data-flagg i hver karakter. Noen l-karakterer er reservert for link-kontroll. De blir brukt til oppstart, flytkontroll og nedtak av linken, og er ikke synlig for brukeren.



Figur 3.4: HIC-karakter 8B/12B DC-koding: Start og Stop bit’et er lagt til for å justere klokka, OP er for paritet og I er et inverteringsbit for å ivareta DC-balansen. d[0]-d[7] er databit’ene, der d[7] er det mest signifikante bit’et.

3.4 BULLIT

BULLIT er BULL’s implementasjon av HS-linken, og er laget for å forsøke å oppfylle alle kravene om hastighet, kosteffektivitet og skalerbarhet (nevnt i innledningen av dette kapitlet). BULLIT består av to 1 Gbaud avsender (transmitter) og mottager (receiver) par med lavnivåprotokoller og fifo-buffere. Avsender har en parallell-til-seriell-konverter og mottager har en seriell-til-parallell-konverter. I dette avsnittet er det gitt en beskrivelse av BULLIT. Først litt generelt og så et par ting spesifikt for det ene sender og mottager settet. For mere informasjon se [BULLIT 94].

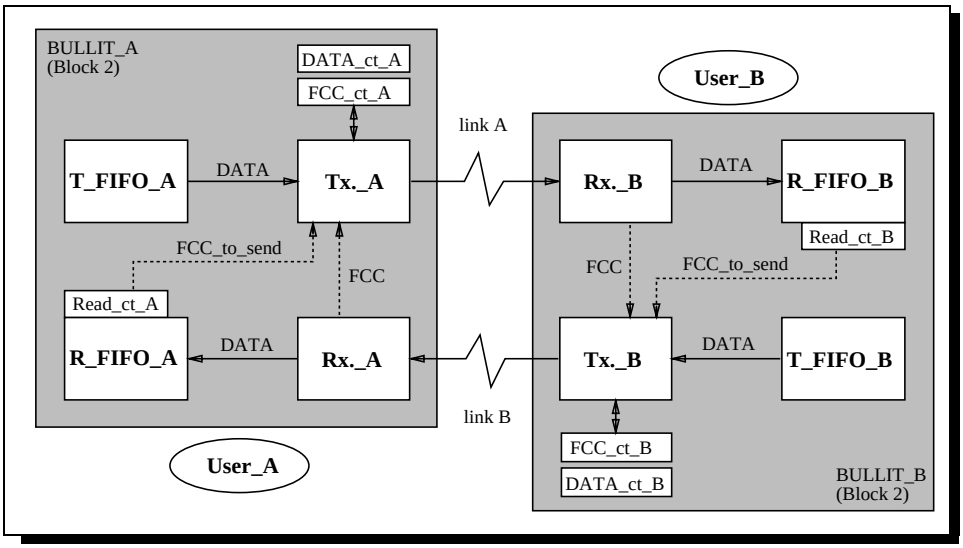
3.4.1 Generelt

BULLIT inneholder to deler, Block 1 og Block 2. Hver del inneholder et avsender/mottager par med både input- og output-fifo-buffere. Hver del gir muligheten for full dupleks toveistransmisjon med en hastighet på den serielle linken fra 700 Mbaud til 1 Gbaud.

Block 2 er en full HS-link implementasjon med flytkontroll og 8B/12B-koding som beskrevet i avsnitt 3.3. Block1 er en toveislink, hvor hverken flytkontroll eller koding er spesifisert. Dette gir brukeren muligheten til å implementere dette selv eksternt. En liten oversikt over Block 1 og Block 2 er gitt i tabell 3.2.

Options	Block 1	Block 2
Link	two-times unidirectional	full bidirectional
Flow Control	No	Yes
Data encoding	Left to the user	8B/12B DC coding
start-up/ shut-down	simple, non-acknowledged	acknowledged
FIFO size	64 Bytes	80 Bytes
user interface	1 Byte	2 Bytes

Tabell 3.2: Egenskaper ved BULLIT’s Block 1 og Block 2. Hentet fra [BULLIT 94].



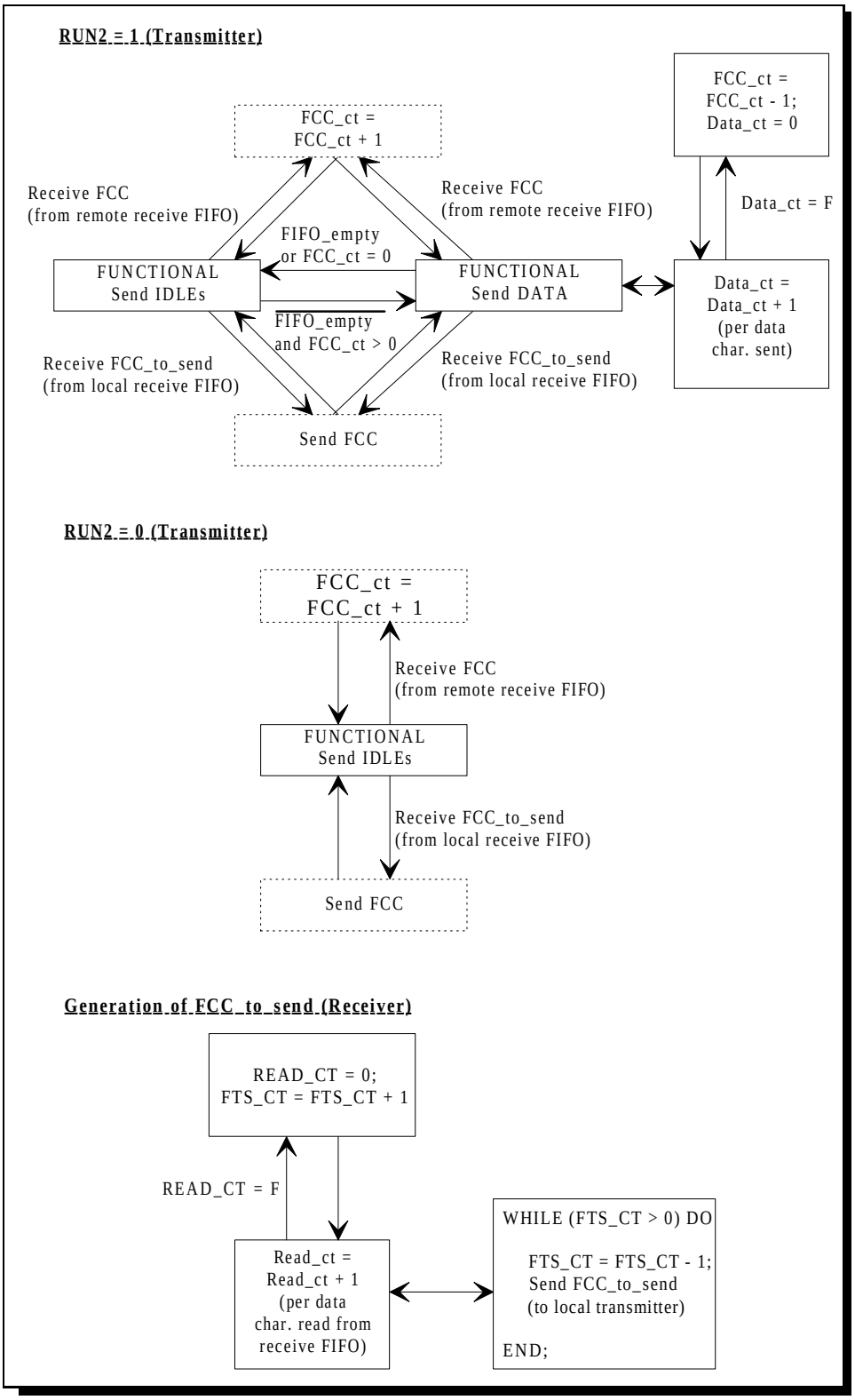
Figur 3.5: BULLIT's Block 2. Kommunikasjon mellom to noder. Hver node har en avsender (Tx.) og en mottager (Rx.) med tilhørende fifo (T- og R-FIFO). Hentet fra [BULLIT 94].

3.4.2 Block 2

I denne oppgaven er det tatt utgangspunkt i BULLIT's Block 2 for å modellere HIC-interfacet til SCI-nodene, siden Block 2 gir en fullverdig protokoll. En skjematisk fremstilling av Block 2 er gitt i figur 3.5. Fifo'ene er på 80 bytes hver. Block 2 bruker den fullt bidireksjonale linken, og har følgende integrerte lavnivå protokoll-funksjoner:

- Kontrollert oppstart og nedtak av de bidireksjonale linkene.
- Automatisk innsetting av IDLE-karakterer når transmitter ikke sender fra T_FIFO eller sender l-karakterer.
- Flytkontrollmekanisme (FCC) for å unngå at fifo'ene blir overfylt.
- 8B/12B DC-balansert kodemønster som gir 256 data- og 126 kontroll-karakterer.

Som beskrevet i avsnitt 3.2.3 sender BULLIT n- og l-karakterer. Flytkontrollen sikrer at mottager-fifo'en (R-FIFO) ikke blir full og overskrevet, ved å bare sende 32 byte (F=32) av gangen, når det er plass til dem i R-FIFO. Flytkontrollen, en FCC-karakter, blir derfor sendt for hver 32. data-karakter. Dette utgjør $\frac{1}{33}$ eller ca 3% av tilgjengelig båndbredde. BULLIT's flytkontrollmekanisme er vist i figur 3.6.



Figur 3.6: BULLIT's flytkontroll. Data-ct teller antall karakterer som blir sendt fra T-FIFO. FCC-ct minker hver gang Data-ct=F. Når FCC-ct=0 må T-FIFO vente til det er mottat en FCC fra mottageren. Read-ct teller antall karakterer som blir tatt ut av R-FIFO. Når Read-ct=F blir det sendt en FCC. RUN2=0 blir brukt ved oppstart, og er ikke brukt i simulatoren. Hentet fra [BULLIT 94].

4

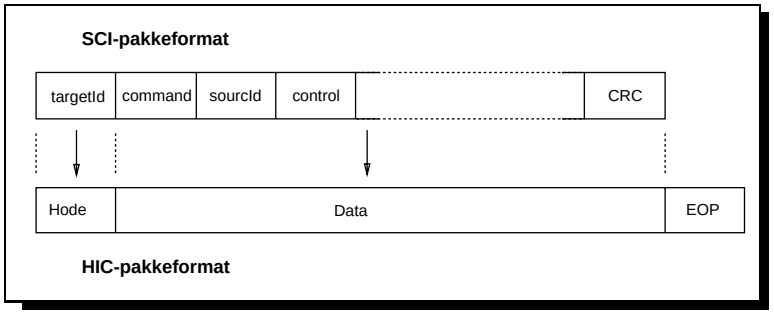
HIC som transportmedium for SCI

Hensikten med oppgaven er å se på systemer hvor man bruker HIC's HS-linker som transportmedium for SCI. Dolphin's GaAs NodeChip utgjør SCI-noden, og BULLIT er brukt som modell for HIC-interfacet.

I dette kapitlet er det sett på noen aspekter ved bruken av HIC. Hvilke tanker som ligger bak, hvilke problemer som oppstår og hvilke valg som er gjort for simulatoren. Det blir beskrevet hvordan SCI's pakkeformat er overført til HIC-format, og hva slags adressering som blir brukt. Problemer med deadlock som følge av delte transaksjoner, blir også diskutert.

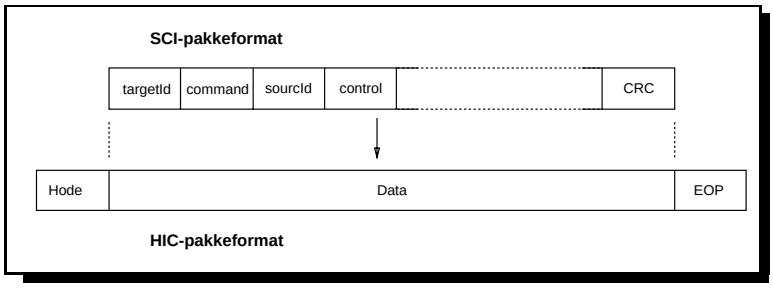
4.1 Pakkeformat og adressering

For å kunne sende SCI-pakkene på et HIC-nett, må pakkeformatet bli omgjort til HIC-format, og man må ha et hode (header) med HIC-adresse som kan ta seg av rutingen på HIC-nettet. En måte å gjøre dette på, og som er brukt i simuleringene i denne oppgaven, er å bruke samme adressen for HIC som for SCI (targetId). På denne måten får man en global SCI-adresse for hele systemet, inklusive HIC-delen. Derfor er dette en grei løsning for små nettverk. Resten av SCI-pakka utgjør da datafeltet i HIC-pakka. Til slutt er det lagt til en EOP-karakter (figur 4.1).



Figur 4.1: Fra SCI- til HIC-pakkeformat slik det er gjort i denne oppgaven.

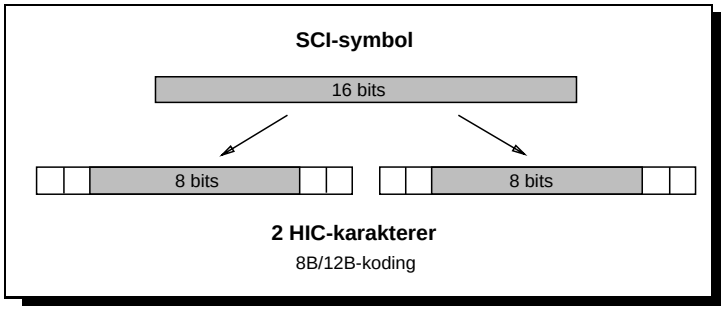
Alternativt kunne man velge å legge på et eget HIC-hode (figur 4.2) som tar seg av rutingen på HIC-nettet og blir strippet vekk etterpå. For store nettverk gir dette muligheten til å ha flere hoder som kan bli strippet vekk underveis (header deletion) [Inmos 93].



Figur 4.2: Fra SCI- til HIC-pakkeformat med eget HIC-hode.

SCI's echo-pakker blir ikke sendt på HIC. Det vil kun bli gitt et lokalt echo slik det er illustrert for fjerntransaksjoner i figur 2.2, hvor HIC vil utgjøre agenten.

På karakter-nivå ser det ut som i figur 4.3. Ett SCI-symbol blir overført til to HIC-karakterer med 8B/12B-koding.



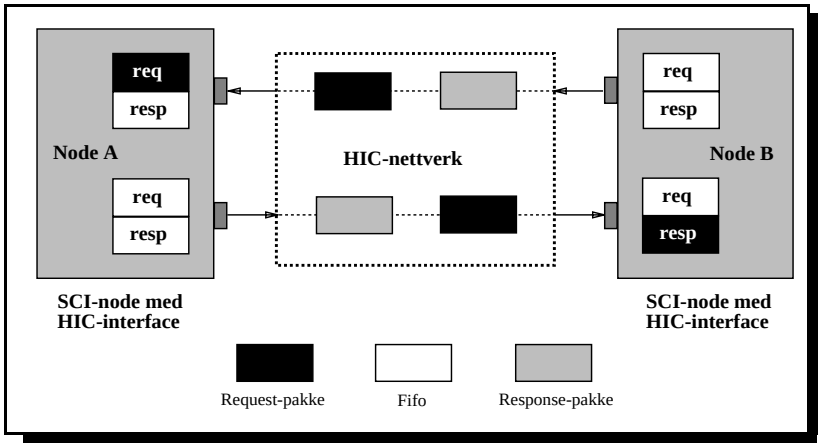
Figur 4.3: Fra ett SCI-symbol til to HIC-karakterer.

4.2 Deadlock

Et viktig problem man må ta hensyn til når det gjelder kommunikasjonsnettverk, er muligheten for deadlock [Tanenb 89, NiMcKi 93, Inmos 93, Hwang 93]. Deadlock er en situasjon hvor nettverket er låst. Det vil si et sett pakker er blokkert og vil aldri nå frem til mottagernoden.

Det er hovedsaklig to grunner til at det oppstår deadlock. Den ene grunnen er som et resultat av nettverkstopologien og rutingalgoritmen som er brukt. Dette blir beskrevet nærmere i kapittel 5. Den andre grunnen er som en følge av delte transaksjoner i et multiprosessor-system med punkt-til-punkt-linker. For eksempel slik som SCI. Request-pakker på vei til en node kan fylle opp og sperre for response-pakker, slik at noden blir stående og vente på en response som aldri kommer frem.

SCI-standarden er laget slik at den tar hensyn til deadlock, ved å bruke separate køer for requester og responser. Men når man bruker et annet transportmedium, slik som HIC er brukt i denne oppgaven, kan det oppstå problemer. En mulig deadlock-situasjon er vist i figur 4.4. Node A venter på en response fra node B, og node B venter på en response fra node A. Request-fifo'ene i begge nodene er opptatt, og request-pakkene i HIC-nettet blokkerer for response-pakkene node A og node B venter på.



Figur 4.4: Eksempel på mulig deadlock ved bruk av enkle linker. Response-pakkene er blokkert av request-pakkene.

4.2.1 Move-pakker

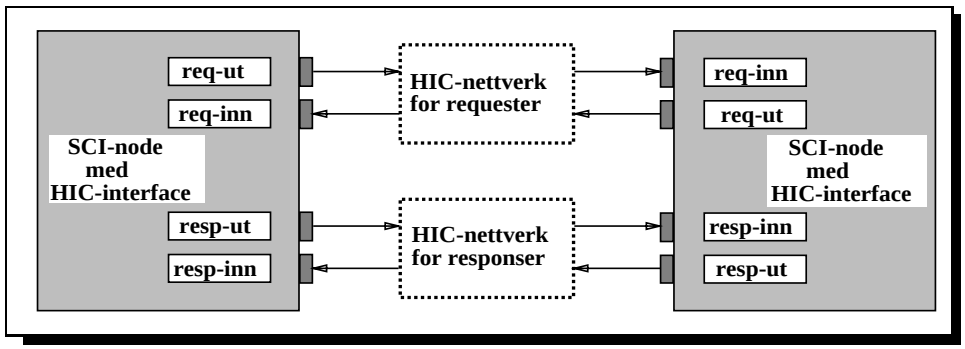
SCI's «move»-pakker resulterer ikke i response-pakker og er derfor deadlock-frie. Ved å bruke bare move-pakker på HIC delen, vil man unngå deadlock. Man vil da kun få meldingsutveksling (message passing), og vil ikke kunne benytte alle SCI's egenskaper. Derfor er dette ikke interessant i denne oppgaven.

4.2.2 Doble linker

Ved bruk av HIC som i denne oppgaven, er det valgt ha to parallelle nettverk, ett for requester og ett for responser, slik det er illustrert i figur 4.5. Dette er en trygg løsning som vil være deadlock-fri, og hele problemet med blokkering av response-pakker er unngått. Doble linker er også brukt i [Bothner 94].

4.2.3 Retry

Enkle linker vil kunne føre til deadlock som beskrevet over. En måte å løse deadlock ved enkle linker på, er å fjerne request-pakker etter et visst antall «retry» eller sykler. Det er ikke sett på retry i denne oppgaven, men måter å gjøre dette på er diskutert i [James 94]. Det man oppnår er mindre hardware-kompleksitet i nettverket siden man kan sende requester og responser på det samme nettet. Det vil derimot være nødvendig med ekstra hardware i HIC-interfacene [James 94].



Figur 4.5: Et SCI-system slik det er tenkt med bruk av doble HIC-linker. Her vil det ikke kunne oppstå en deadlock-situasjon som i systemet i figur 4.4.

5

HIC-svitsj

Det var ikke laget noen svitsjer for HIC's HS-link da oppgaven ble påbegynt, men det var nødvendig å modellere en HIC-svitsj for bruk i simulatoren. Det ble derfor tatt utgangspunkt i ST C104 [ST C104 94], siden denne er laget for HIC's DS-link. Det er valgt å la svitsjene være like for både request- og response-nettverket.

Dette kapitlet beskriver hvordan HIC-svitsjen brukt i simulatoren er modellert, hvilke rutingalgoritmer som er valgt, og hvilke hensyn som er tatt i forhold til deadlock. Det er også sett på strategier og algoritmer for å øke nettverksytelsen. Til slutt er det gitt en oppsumering.

5.1 Wormhole-ruting

I en del pakkesvitsj-nettverk tar hver svitsj inn hele pakka, dekoder rutinginformasjonen og sender pakka videre til neste node (store & forward). Dette krever lagringsplass i hver svitsj og fører til forsinkelser mellom mottak og videresending.

Med «wormhole»-ruting unngår man disse problemene [NiMcKi 93, Hwang 93, Inmos 93]. En pakke blir delt opp i et visst antall enheter (flits), som for HIC vil være 8B/12B-karakterer. Hvor pakka blir sendt videre, blir bestemt med en gang hodet på pakka er lest. Hvis utgangen er ledig, blir hodet sendt ut og resten av enhetene i pakka følger etter uten forsinkelser. Hvis utgangen er i bruk, må hodet vente, og flytkontrollen sørger for at resten av pakka blir lagret i buffere langs rutingveien. Hodet på pakka lager en temporær krets som lukker seg etter at pakka har passert. Navnet wormhole-ruting henspeiler på en mark (worm) som kryper gjennom sand. Marken lager et hull som lukker seg igjen etter den. Wormhole-ruting fører til at en pakke kan gå gjennom flere svitsjer samtidig, og hodet på pakka kan komme fram til mottageren før hele pakka er sendt fra avsender. Slik blir det minimalt med forsinkelser.

5.2 Intervall-ruting

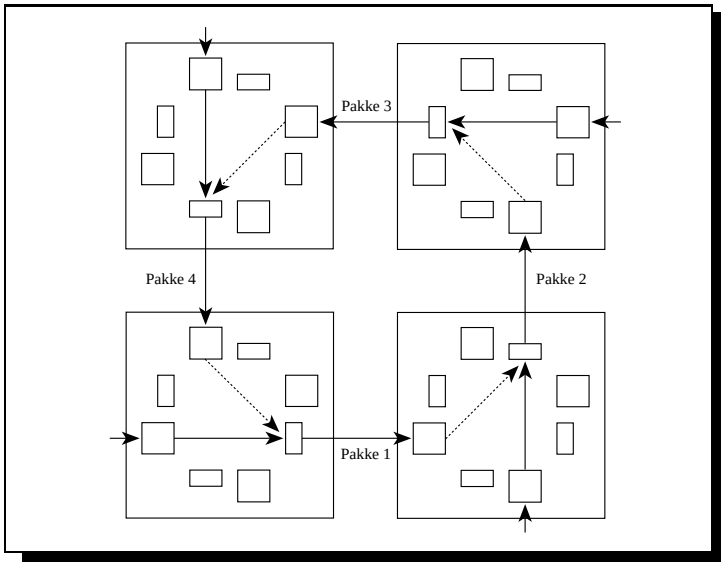
Wormhole-ruting trenger en effektiv rutingstrategi for å velge hvilken utgang en pakke skal til. Intervall-ruting er en egnet strategi, hvor hver utgang blir gitt et visst intervall [LeeuTan 87, Inmos 93]. Intervallet inneholder numrene på alle terminalnoder man kan nå fra den utgangen. Når en pakke ankommer svitsjen, blir utgangen

valgt ved å sammenligne adressefeltet i hodet med intervall-settene. Intervallene er kontinuerlige og ikke-overlappende, slik at hvert hode kun hører til ett av intervallene.

Det er mulig å sette labler i de fleste vanlige topologier, som matrise-nettverk, trær og multi-stage-nett[1], slik at pakkene følger en optimal vei, og nettverket blir deadlock-fritt. Noen topologier, for eksempel ringer og en del k-ary n-cubes, kan ikke uten videre få labler optimalt og deadlock-fritt, men det er mulig å få det til med noen ekstra linker. Intervall-rutingen sørger for at hver pakke tar korteste vei og når sitt mål. En overføring av en pakke fra en link til en annen påvirker ikke en annen pakke som blir sendt mellom to andre linker.

5.3 Deadlock og rutingalgoritmer

Som nevnt i kapittel 4.2, kan deadlock oppstå som følge av en uheldig kombinasjon av topologi og rutingalgoritme. Her følger en beskrivelse av noen algoritmer det er sett på i forbindelse med simulatoren.



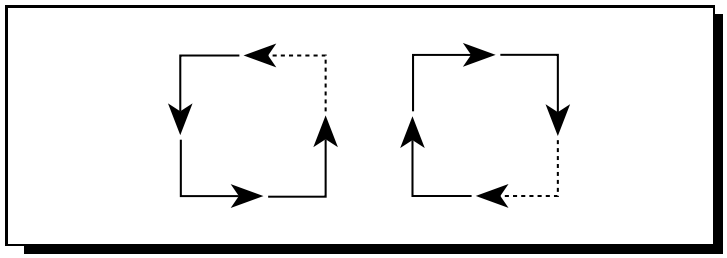
Figur 5.1: Deadlock mellom fire svitsjer og fire pakker. Pakkene blir stående å vente på hverandre i en sykel.

Deadlock i wormhole-ruting oppstår når pakker blir stående å vente på hverandre i en sykel, slik det er illustrert i figur 5.1. Fire pakker på vei i forskjellige retninger prøver alle å «svinge» til venstre og blir stående å vente på hverandre. Hvis bare én av pakkene hadde svingt en annen vei, hadde deadlock vært unngått. Multi-stage-nett er deadlock-frie ettersom de ikke har sykler. Deadlock kan imidlertid oppstå i matrise-nettverk og k-ary n-cubes [Dally 87].

[1] Topologiene simulert i denne oppgaven er vist i figur 6.5 i kapittel 6. Ellers er de fleste nettverkstopologier beskrevet i [Hwang 93].

5.3.1 West-first-ruting

Det er vist at for n-dimensjonale matrise-nettverk, kan man fjerne en fjerdedel av svingene i en sykel for å forhindre deadlock [GlassNi 92]. Ved å analysere hvilke 90°-svinger som kan oppstå mellom to retninger i en matrise, og hvilke sykler disse danner, kan man fjerne en sving i hver sykel. En slik algoritme som også passer sammen med intervall-ruting, er «west-first»-ruting [GlassNi 92, NiMcKi 93]. Figur 5.2 illustrerer syklene i en 2-dimensjonal matrise. West-first-ruting gjør at de stiplete svingene er ulovlige.



Figur 5.2: Syklene i en 2-dimensjonal matrise. Hvis nord er oppover på figuren, gjør West-first-ruting de stiplete svingene ulovlige.

5.3.2 Random-ruting

I et nettverk hender det dessverre at enkelte linker blir mere brukt enn andre. En slik link som stadig er opptatt, blir fort en flaskehals. Det kan føre til at pakker må vente, og det oppstår forsinkelser. For å løse dette, kan man midlertidig sende pakka til en annen tilfeldig valgt svitsj, og så blir pakka sendt derfra videre til det opprinnelige målet. Denne algoritmen er gjerne kalt «random»-ruting og er laget for å øke kapasiteten og minske forsinkelsene ved høy belastning [Valiant 82, Inmos 93, Tanenb 89][2]. Måten dette blir gjort på er at det blir lagt på et ekstra hode med den nye midlertidige adressen. Det ekstra hodet blir så strippet vekk når pakka ankommer den midlertidige mottagersvitsjen.

En svakhet kan være randomfunksjonen. Spredningen av trafikken er avhengig av hvor god randomfunksjonen er. Random-ruting fører også til ruting i to faser i det samme nettet, noe som gir mer kompliserte rutingveier. Dette kan i verste fall føre til deadlock [KimDas 94]. [Inmos 93] foreslår dette løst ved å bruke et separat nettverk for random-rutingen. Random-ruting er ikke implementert i denne oppgaven.

5.3.3 Gruppering av linker

For å øke båndbredden og minske forsinkelsen, kan man gruppere flere linker mellom to svitsjer. [ST C104 94] kaller dette «grouped adaptive»-ruting:

Når en innkommende pakke blir rutet mot den første linken i en gruppe, blir den egentlig rutet mot en hvilken som helst link i gruppen. Enhver ledig link i gruppen kan ta i mot og sende pakka. Kun én link vil ta imot og sende pakka hvis to eller

[2] Kalt «Universal routing» i [Inmos 93] og «Random walk» i [Tanenb 89].

flere linker er ledige eller blir det samtidig. Hvis flere pakker blir rutet til en gruppe enn det er output-linker, vil de «overskridene» input-linkene som venter på å sende, bli lagret. En av de ventende input-linkene får aksess til en output-link så snart en output-link i gruppen blir ledig.

For å begrense oppgavens omfang, er det valgt å ikke ta med gruppering av linker. Det er imidlertid en ting det kunne vært interessant å se på senere.

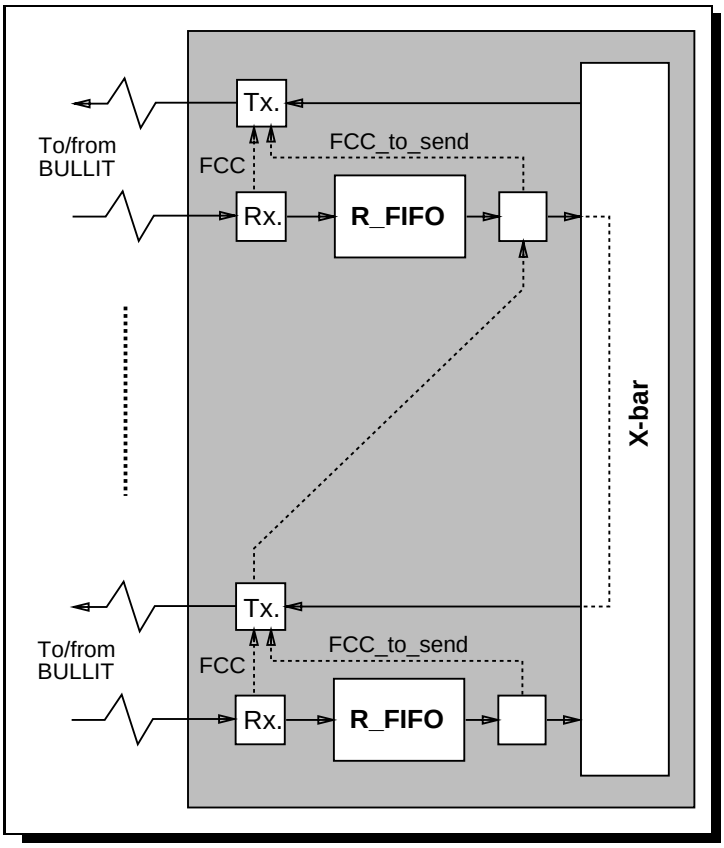
5.4 Oppsummering

Med utgangspunkt i ST C104 er det utviklet en HIC-svitsj for simuleringene av HS-link-systemene i denne oppgaven.

Siden BULLIT (kapittel 3.4) utgjør HIC-interfacet i SCI-nodene, er det naturlig og også bruke BULLIT og BULLIT's protokoll ved modellering av HIC-svitsjen. Svitsjen har derfor en 80 bytes mottager-fifo (R-FIFO) for hver input-link. Denne fungerer også som avsender-fifo (jmfr. BULLIT's T-FIFO). Flytkontrollen fungerer som for BULLIT, ved at en FCC blir sendt når det er plass til å motta F=32 nye karakterer. Kreditt-tellere (FCC-ct og Read-ct) holder orden på hvor mange som er sendt.

Ellers er svitsjen laget med hensyn på de egenskaper fra ST C104 som tidligere er nevnt i dette kapitlet: wormhole- og intervall-ruting. Svitsjen har en «crossbar» (X-bar) som kan rute en pakke fra en hvilken som helst input-link til en hvilken som helst output-link. En overføring mellom to linker påvirker ikke dataraten eller forsinker en annen pakke som samtidig blir overført mellom to andre linker. Hvis en input-link ønsker å sende til en output-link som er opptatt, må den vente i en rettferdig kø. Linken som har ventet lengst, er den første som får sende når output-linken blir ledig (first come, first serve). En link får ikke sende to pakker etter hverandre til samme link hvis en annen link venter. En skisse av svitsjen er gitt i figur 5.3.

OMI/HIC-prosjektet har i det siste startet utviklingen av en HIC-svitsj for HS-link de har kalt RCUBE (Rapid Reconfigurable Router) [RCUBE 94]. Denne er ganske lik svitsjen skissert her. Total forsinkelse gjennom svitsjen på 84 ns brukt i denne oppgaven, er hentet fra arbeidet med RCUBE.

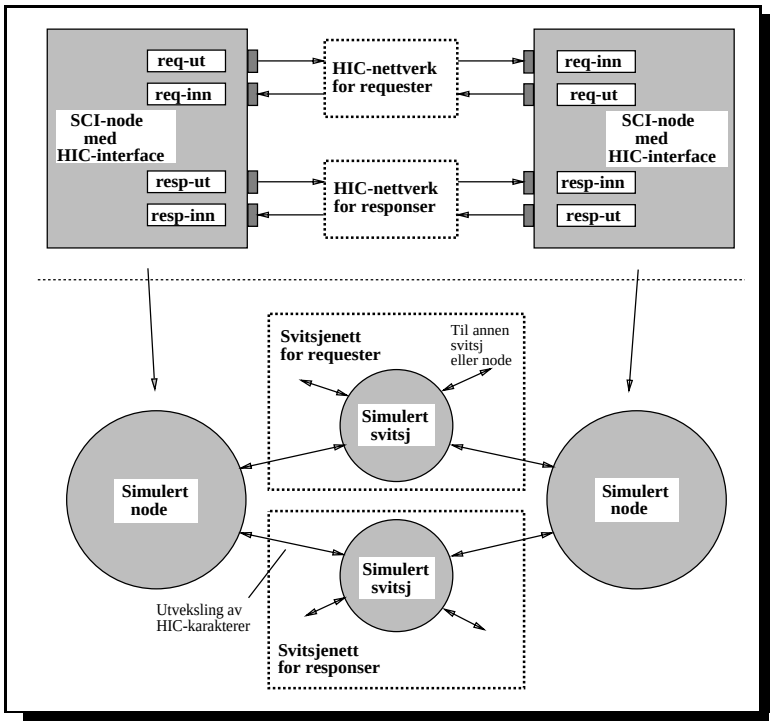


Figur 5.3: Skisse av HIC-svitsjen slik den er modellert i denne oppgaven. På figuren sender øverste input- til nederste output-link. Øverste R-FIFO fungerer da som mottager-fifo for input-linken og avsender-fifo for output-linken. Flytkontrollen fungerer som for BULLIT.

6

Konstruksjon av simulatoren

I de foregående kapitlene er det beskrevet hvordan man kan sende SCI-pakker på HIC-nettverk. Dette kapitlet beskriver hvordan simulatoren er konstruert, dens forskjellige moduler og funksjoner, og hvordan den virker. I figur 6.1 er det illustrert hvordan systemet fra figur 4.5 er overført til modellen i simulatoren.



Figur 6.1: Fra referent til simulert modell. Figuren illustrerer hvordan systemet vist i figur 4.5 er modellert i simulatoren. HIC-nettverkene består av svitsjer og linker.

Videre er det forklart hvordan intervall-ruting er implementert, og hvordan flytkontrollen fungerer i simulatoren. De forskjellige parameterene for nettverksbelastning, forsinkelser og simuleringstid er beskrevet, og det er forklart hvordan innhenting og beregning av statistikk foregår. Det er også forklart hvordan nettverkstopo-

logiene er konstruert, og hvordan syntaksen på topologi- og labelfilene skal være. Til slutt kommer litt om randomfunksjonene og hvordan de er brukt.

Simulatoren inneholder noder som sender pakker til hverandre over et nettverk av en eller flere svitsjer. En node representerer en SCI-node med prosessor og hukommelse som genererer request-pakker og response-pakker som følge av disse. Request-pakkene er adressert tilfeldig og blir generert ved tilfeldige tidsintervaller. Ved å variere hvor ofte en pakke blir generert, kan man variere nettverksbelastningen.

Kort fortalt virker simulatoren slik: Den leser først inn en topologifil som beskriver hva slags nettverk som vil bli simulert. Med andre ord forteller fila simulatoren hvordan noder og svitsjer henger sammen. Når nettverket er konstruert, starter simulatoren og går dét antall klokkesykler som er spesifisert. En klokkesykel i simulatoren tilsvarer 2 ns. Under hele simuleringen blir forskjellige tellere inkrementert. Ønskede statistiske data blir beregnet og skrevet ut på en resultatfil når simulatoren er ferdig.

Simulatoren er skrevet i programmeringsspråket C++. Dette er objektorientert og egner seg derfor spesielt godt til å beskrive et nettverk som består av noder, svitsjer og linker mellom disse. Mer om C++ i appendiks B.

6.1 Klasser og funksjoner

Her blir datastrukturene presentert. På øverste nivå er klassene *Node*[1] og *Switch* som representerer de fysiske enhetene node og svitsj i nettverket. *Node* inneholder funksjonen *packet_handler()* som tar seg av generering av nye request-pakker, mottak av requester og generering og mottak av responser. *Switch* inneholder også en *packet_handler()*-funksjon som ruter pakkene til riktig utgang.

Hver av utgangene og inngangene (input- og output-linkene) til nodene og svitsjene er representert ved objekter av klassen *HIC_interface* som tar seg av HIC/BULLIT-protokollen. *HIC_interface* har igjen subklassene *Node_face* og *Switch_face* for henholdsvis noder og svitsjer, ettersom disse er forskjellige, men har noe felles (figur 6.2).

Pakkene er bygget opp av objekter av klassen *HICchar*. *HICchar* representerer en karakter (character) som blir sendt mellom nodene og svitsjene. *HICchar*-objektene inneholder variable som forteller om det er en n-karakter (data eller EOP) eller en l-karakter (FCC).

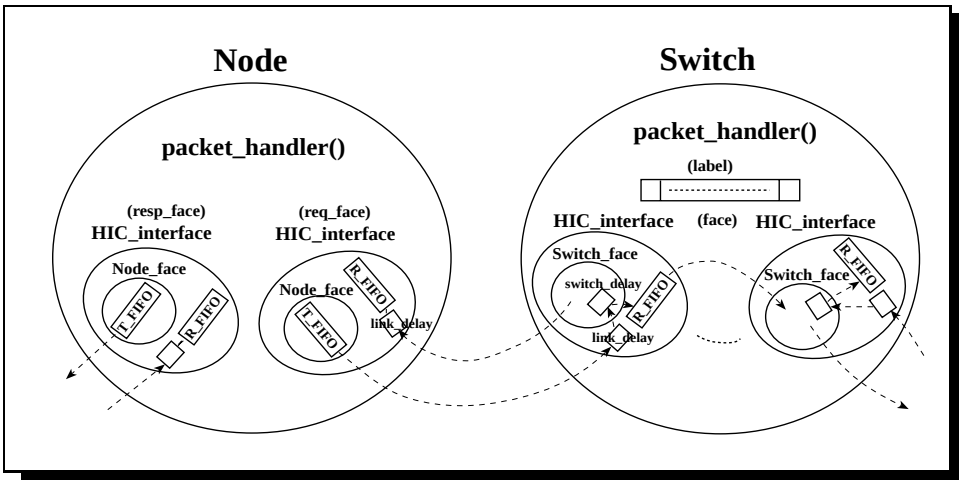
Simulatoren tar ikke hensyn til signalforskyvning og støy på de fysiske linkene.

6.1.1 Nodene

Hver node representerer i utgangspunktet bare én enkel SCI-node med prosessor, hukommelse og to HIC-interface som er modellert etter BULLIT (kapittel 3.4). Ved å endre belastningsvariablene kan man godt la en node representere en hel SCI-ring eller et større SCI-nett, for eksempel en k-ary n-cube[2]. Likevel er det som skjer på SCI-siden kun representert ved parametere som gir riktige forsinkelser (delay). Det som blir simulert, er derfor bare det som skjer på HIC-nettet.

[1] Navn som refererer til variabler, funksjoner eller klasser i programmet, er skrevet i *kursiv*.

[2] Nettverkstopologi simulert i [HulBot 93], også beskrevet i [Hwang 93].



Figur 6.2: Skisse av objektene Node og Switch.

Klassen *Node* inneholder:

- *req_face* og *resp_face* – objekter av klassen *Node_face/HIC_interface* (avsnitt 6.1.3) en for request-nettet og en for response-nettet.
- Funksjonen *packet_handler()*.
- Funksjonen *make_packet()*.
- Variable som holder orden på når requester og responser blir sendt og hvor mange utestående requester noden har.
- Variable for innhenting av statistikk.

Hovedfunksjonen i nodene er funksjonen *packet_handler()*. Den tar seg av generering og mottak av pakker. Ved visse tidsintervaller (avsnitt 6.4) blir det generert request-pakker ved hjelp av funksjonen *make_packet()*. Disse blir puttet i *T_FIFO* for requester. Når så en request kommer inn fra *R_FIFO*, leser mottager avsenderadressen og genererer en response-pakke. Det tar så 100 sykler (200 ns)[3] før response-pakka blir sendt til *T_FIFO* for responser. Når en response kommer inn, blir det trukket en ny tid for når neste request blir sendt. *packet_handler()* sørger også for å beregne en del statistikk (avsnitt 6.5).

6.1.2 Svitsjene

Klassen *Switch* er laget ut fra svitsjen i kapittel 5, med wormhole-ruting og intervallruting. *HIC_interface*-objektene tar seg av HIC/BULLIT-protokollen og kommunikasjonen med nodene.

[3] Verdien er valgt ut fra hvor lang tid en response i en SCI-hukommelse vil ta. [Bugge 90] bruker 240 ns, men nye og raskere hukommelser tilsier at 200 ns er et rimelig tall. Også brukt i [HulBot 93].

Klassen *Switch* inneholder:

- Variabelen *size*, som bestemmer størrelsen.
- Funksjonen *packet_handler()*.
- Funksjonen *find_label()*.
- Array, *face*, med objekter av klassen *Switch_face/HIC_interface* (kapittel 6.1.3).
- Array, *label*, som inneholder label-verdiene til hvert *Switch_face/HIC_interface*.

Størrelsen på arrayene avhenger av størrelsen på svitsjene. I denne oppgaven er det gjort simuleringer av 4x4-, 8x8- og 16x16-svitsjer. Størrelsen blir satt i topologifila (avsnitt 6.6). *Switch* har også funksjonen *packet_handler()*, men denne er forskjellig fra den i *Node*. *packet_handler()* håndterer rutingen. Det vil si at den finner adressen en pakke skal til og plukker ut riktig utgang (*Switch_face/HIC_interface*) ut fra verdiene i label-arrayen. Dette blir gjort ved hjelp av funksjonen *find_label()* (avsnitt 6.2).

6.1.3 Klassen *HIC_interface*

Klassene *Node* og *Switch* inneholder som nevnt *HIC_interface*-objekter som tar seg av HIC/BULLIT-protokollen. *HIC_interface*-objektene er forskjellige i *Node* og *Switch*, men har en del felles. Dette er derfor greit ordnet ved hjelp av subklasser. De forskjellige delene er modellert ut fra HIC og BULLIT som beskrevet i kapittel 3. Se også figur 3.5.

Felles, klassen *HIC_interface*:

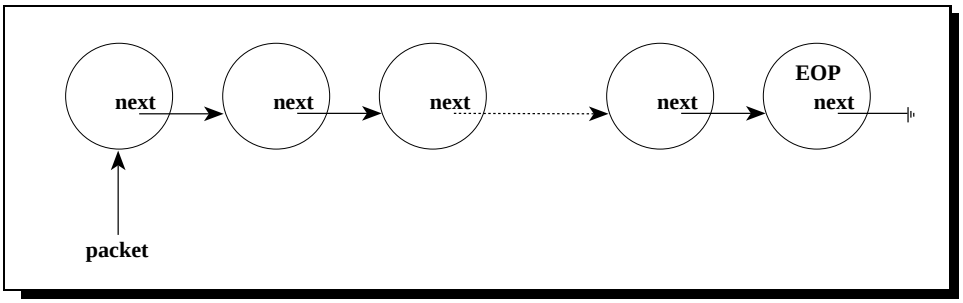
- *link_delay* – objekt av klassen *Delay* med parameter *LINK_DELAY*.
- *R_FIFO* – objekt av klassen *Fifo*. Representerer mottager-fifo.
- *face_link* – peker til *HIC_interface* i *Node* eller *Switch* (output-link).
- Variabler som holder orden på sending og mottak av FCC, bl.a. *FCC_ct*, *Read_ct*.

I *Node*, klassen *Node_face*:

- *T_FIFO* – objekt av klassen *Fifo*. Representerer avsender-fifo.
- *transmitter()* – (avsnitt 6.1.5).
- *receiver()* – (avsnitt 6.1.6).

I *Switch*, klassen *Switch_face*:

- *transmitter()* – (avsnitt 6.1.5).
- *receiver()* – (avsnitt 6.1.6).
- *waiting_faces* – objekt av klassen *Switch_face_fifo*. En vente-fifo som blir brukt hvis output-linken er opptatt. Laget på samme måte som klassen *Fifo*.
- *switch_delay* – objekt av klassen *Delay* med parameter *SWITCH_DELAY*.
- Variabler som blant annet holder orden på hvilket *HIC_interface* som sender, når en ny pakke kommer og mottageradresser.



Figur 6.3: Pakke bestående av en kjede med HICchar-objekter. I simulatoren består pakkene av 81 karakterer.

6.1.4 Klassen *HICchar*

Det blir sendt to typer pakker på HIC-nettet, requester og responser. I simulatoren er request- og response-pakkene gjort like. Lengden på begge SCI-pakkene er satt til 80 bytes som i [HulBot 93], det vil si 64 bytes data. Overført til HIC som beskrevet i kapittel 4, blir det 81 8B/12B-karakterer, inklusive EOP.

For å modellere disse n-karakterene som utgjør en pakke, og l-karakterene som beskrevet i kapittel 3, er det laget en klasse kalt *HICchar*. En skisse av hvordan en pakke er bygget opp er vist i figur 6.3.

Klassen *HICchar* inneholder variablene og pekerne:

- *next* – peker til neste *HICchar* i pakka.
- *fifo_next* – peker til neste *HICchar* i en fifo.
- *data* – blir blant annet brukt til statistikk.
- *control_ID* – blir brukt til å skille n-karakterer (vanlig data og EOP) og l-karakterer (FCC).
- *address* – inneholder adressen til mottagernoden i 1. og 2. *HICchar*, og adressen til avsendernoden i 5. og 6. *HICchar* i en pakke.

6.1.5 Funksjonen *transmitter()*

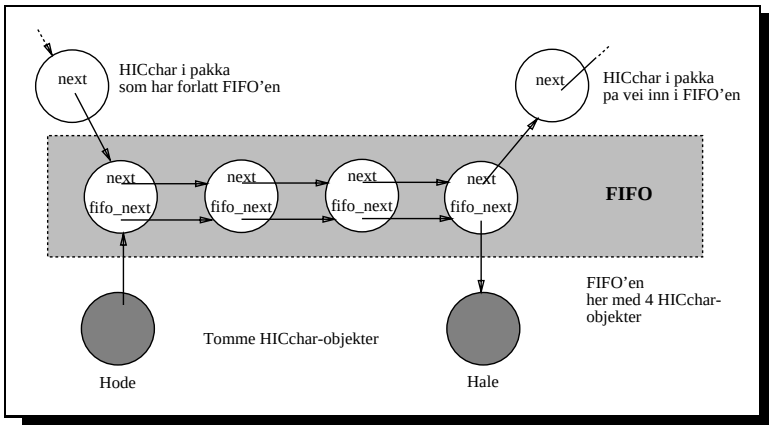
Funksjonen *transmitter()* blir kalt hver sykel. Den sender *HICchar*-objekter ut på linken hvis det er noe å sende. Først sjekker den om den skal sende FCC, hvis ikke sjekker den om den skal sende noe fra fifo'en (*T_FIFO* i *Node_face* og *R_FIFO* i det *HIC_interface* som sender i *Switch_face*).

I *Switch_face* sørger *transmitter()* også for at linken er opptatt i riktig antall sykler for hver pakke eller FCC den sender. Når *transmitter()* har sendt EOP, sjekker den *waiting_faces* om det er andre som ønsker å sende til samme linken, og sørger eventuelt for at neste i køen får sende.

6.1.6 Funksjonen *receiver()*

Tilsvarende funksjonen *transmitter()* blir også *receiver()* kalt hver sykel. Den tar i mot *HICchar*-objekter fra linken. Hvis det er noen, sjekker den *control_ID* om det er FCC, vanlig data eller EOP. Hvis det er FCC, oppdaterer den FCC_ct. Hvis det er vanlig data, putter den *HICchar*-objektet i *R_FIFO*.

I *Switch_face* fører EOP til at *receiver()* forbereder seg på at neste *HICchar* tilhører en ny pakke, og adressefeltet vil bli lest. For å få riktig totalforsinkelse gjennom svitsjen, blir *HICchar*-objektene først sendt gjennom *switch_delay* før de havner i *R_FIFO*.



Figur 6.4: Fifo med fire HICchar-objekter. Hodet peker på halen når fifo'en er tom.

6.1.7 Klassen *Fifo*

I avsnitt 6.1.3 går det frem at hvert objekt av klassen *HIC_interface* har fifo'er. En fifo er en først-inn-først-ut kø. Første objekt som kommer inn i køen er første som går ut. HIC/BULLIT's flytkontroll sørger for at en fifo aldri blir overfylt, og holder orden på om et *HICchar*-objekt skal bli sendt videre eller bli buffret (kapittel 3). I nodene blir alltid *T_FIFO* fylt helt opp når en ny pakke er generert.

Klassen *Fifo* består av en pekerkjede med *HICchar*-objekter (avsnitt 6.1.4). Som «hode» og «hale» er det laget to tomme *HICchar*-objekter (dummies). Disse angir begynnelse og slutt på fifo'en. Når fifo'en er tom, peker hodet på halen. Sist ankomne *HICchar* er sist i kjeden, og den som har vært der lengst, er først. Hver klokkesykel kan en *HICchar* forlate og/eller entre fifo'en. Forskjellige funksjoner sørger for å putte *HICchar*-objektene inn eller ta de ut. En illustrasjon av fifo'en er gitt i figur 6.4.

6.1.8 Klassen *Delay*

På de fysiske linkene går ikke signalene med uendelig hastighet. Derfor er det laget en klasse, *Delay*, som modellerer forsinkelsen på linkene. Denne klassen blir også brukt til å få riktig totalforsinkelse gjennom svitsjene.

Klassen *Delay* består av en array med pekere til *HICchar*-objekter og funksjoner som putter objekter inn og tar de ut. Lengden på arrayen bestemmer hvor stor forsinkelsen blir. Hver *HICchar* må innom hver peker, det vil si at den rykker en peker frem for hver klokkesykel. Er for eksempel lengden satt til 2 gir det en forsinkelse på 2 klokkesykler.

6.2 Ruting

I svitsjene blir pakkene rutet ved hjelp av intervall-ruting (kapittel 5). En array med lablene ligger parallelt med *Switch_face/HIC_interface*-arrayen. Først blir adressen i innkommende pakke lest i funksjonen *receiver()*, så blir den sammenlignet med lablene ved hjelp av funksjonen *find_label()*. Når man har funnet den minste labelen som er større enn adressen, blir pakken sendt ut på tilhørende *Switch_face/HIC_interface*. På grunn av intervall-rutingen må man ha en unik nettadresse for hver node, også kalt global adresse. Nodene trekker adressen en request-pakke blir sendt til ved hjelp av uniformfordelingen (avsnitt 6.7). Dette gir en jevn distribusjon av pakker i nettet.

6.3 Flytkontroll

Flytkontrollen (FCC) er i simulatoren implementert slik den er beskrevet i kapittel 3. Figur 3.6 gir en god beskrivelse av hvordan flytkontrollen fungerer. En FCC blir sendt som en vanlig *HICchar* med *control_ID=FCC*. Siden det tar litt tid fra en FCC er lest inn til kreditten *FCC_ct* blir oppdatert, og tilsvarende fra *Read_ct=0* til en FCC blir sendt, er det lagt inn en forsinkelse for dette. Begge er satt til 12 ns.

6.4 Variasjon av nettverksbelastning

Simulatoren er laget slik at det er mulig å trekke tiden for hvor ofte requester blir generert med både normal-, negativ-eksponentsial- og uniformfordelingen, som i [HulBot 93]. Resultatene fra [HulBot 93] har imidlertid vist at de forskjellige fordelingene ikke ga noen særlig forskjell på resultatene. Det er i denne oppgaven derfor kun simulert med uniformfordelingen med forskjellige verdier. Minimumsverdi er som i [HulBot 93] satt til 20 ns, og som maksimumsverdi er det simulert med verdiene 200 ns, 400 ns, 1000 ns, 2000 ns, 4000 ns, 10000 ns og 20000 ns. Antall utestående requester er også variert med 1, 2 og 4 som i [HulBot 93]. Tabell 6.1 viser de forskjellige parameterne som er brukt i simulatoren.

Simuleringstiden er forsøkt valgt slik at man får nok pakker gjennom det totale nettet til å få et godt statistisk grunnlag. Som en test ble den største topologien (4x4-matrisen) simulert med minste påtrykk og forskjellige lengder på simuleringstiden. Det viste seg at resultatene ble ganske like ved 100000 sykler eller mer. Derfor er alle simuleringene kjørt med 500000 syklers simuleringstid, tilsvarende 1ms, da dette skulle være lenge nok til å gi godt nok statistisk grunnlag.

Nodene er som i [HulBot 93] ikke gitt noen form for prioritet, men noder og svitsjer blir gjennomgått i samme rekkefølge i hver klokkesykel. De som er først blir med andre ord alltid eksekvert først. Hvis to input-linker ønsker å sende til samme

Parameter	Verdi
SCI pakkelengde	80 bytes
SCI pakkehode	16 bytes
SCI pakkedata	64 bytes
HIC pakkelengde (SCI+EOP)	81 karakterer
Tid å lese én HIC-karakter	12 ns
Tid å lese en hel pakke	972 ns
Antall utestående requester	1, 2 og 4
Forsinkelse over en link	2 ns
Forsinkelse gjennom en svitsj	84 ns
Forsinkelse gjennom et HIC-interface	40 ns
Tid å lage en FCC	12 ns
Tid fra FCC er mottatt til FCC_ct blir oppdatert	12 ns
Min. tid å lage en response	200 ns
Min. forsinkelse fra response til ny request	20 ns
Maks. forsinkelse fra response til ny request	200-20000 ns
Lengde på simulering	1 ms
1 sykel i simulatoren =	2 ns

Tabell 6.1: Parameterverdier for simulatoren[4].

output-link samtidig (i samme sykel), vil den input-linken som blir eksekvert først, få sendt først. Linken som må vente, vil være den neste som får sende, slik at man likevel får et rettferdig kø-system. En link som sender får ikke sende en ny pakke hvis en annen input-link venter.

6.5 Innhenting av statistikk

Hensikten med oppgaven er å se på ytelsen til SCI-systemer med HIC som transportmedium. Derfor er det laget en simulator som kan simulere slike systemer. Under simuleringen sørger diverse tellere for innhenting av statistikk. Når selve simuleringen er ferdig blir ønskede data beregnet og skrevet ut på en resultatfil. Ønskede data er i dette tilfellet båndbredde, latency og trafikk på linkene. Resultatene er presentert i kapittel 7, her er en forklaring på hvordan de er beregnet:

Effektiv total båndbredde (throughput):
Dette er regnet ut ved å telle antall pakker som kommer frem til en node, for det totale nettet, det vil si både request- og response-nettet. For hver pakke er det kun 64 bytes data. Dette er igjen normalisert med hensyn på simuleringstiden. Resultatet er angitt i Mbytes/s.

[4] Ikke-dokumenterte verdier er valgt i samråd med Ernst Kristiansen.

Latency:
Latency er tiden fra en pakke er generert/blir sendt til hele pakka er kommet frem til mottagernoden. Antall pakker er summert opp innenfor intervaller med bredde 100 ns for å få en fordelingskurve. Det er også beregnet total gjennomsnittlig latency pr node ved å summere opp latency for alle pakkene og normalisere på antall noder. Minimum latency er gitt ved pakka som brukte minst tid, og tilsvarende er maksimum latency gitt ved pakka som brukte mest tid. Resultatet er angitt i nanosekunder (ns).

Gjennomsnittlig trafikk på linkene:
Dette er regnet ut ved å telle alle *HICchar*-objektene som blir sendt fra *transmitter()*, og så er dette normalisert med hensyn på totalt antall linker og simuleringstid. Det er også sett på hvor mye av trafikken som er FCC. Resultatet er angitt i Mcharacters/s pr link.

I en simulator blir det gjort en del forenklinger, slik at det alltid vil være noe usikkerhet forbundet med resultatene. Mer om dette i kapittel 7.8.

6.6 Konstruksjon av nettverk

Det er forsøkt å finne en grei måte å representere nettverkstopologiene på uten å måtte skrive et program for hver enkelt. Dette er gjort ved hjelp av to filer for hver topologi. En topologifil forteller hvordan svitsjene henger sammen, og en labelfil angir lablene i hver svitsj. Slik er det også mulig å prøve ut forskjellige måter å sette labler på for samme topologi. RCUBE (omtalt i kapittel 5.4) er en 8x8-svitsj [RCUBE 94]. Derfor er det simulert 8 systemer med 8x8-svitsjer satt sammen i forskjellige nettverkstopologier. Topologinavnene henspiller på plasseringen av svitsjene. Topologien er lik for request- og response-nettet:

a) En enkel svitsj med 8 noder

b) 8-svitsjers multi-stage med 32 noder

c) 8+4-svitsjers indirekte multi-stage med 32 noder

d) 8+2-svitsjers indirekte multi-stage med 48 noder

e) 3x3-matrise med 48 noder

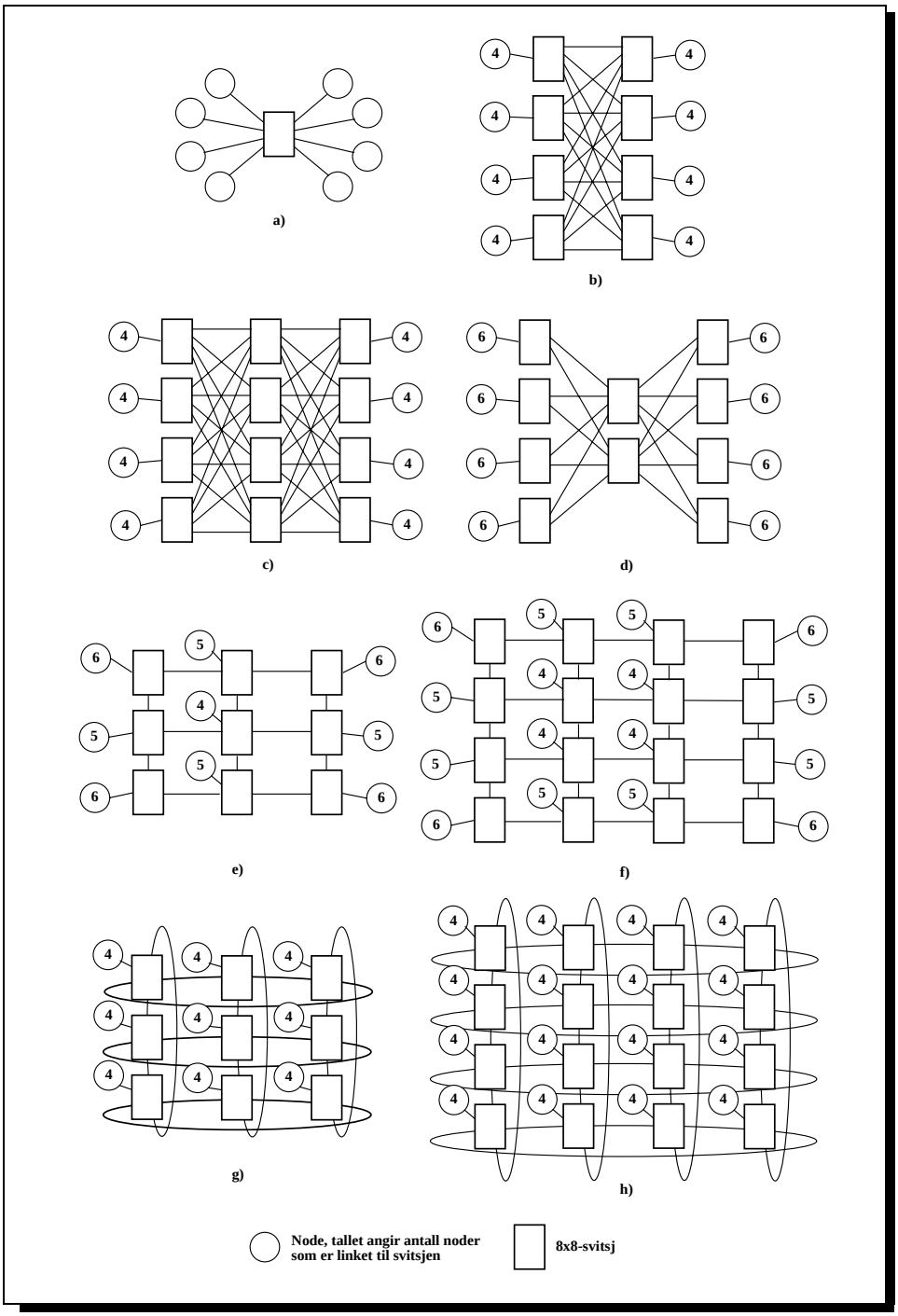
f) 4x4-matrise med 80 noder

g) 3-ary 2-cube med 36 noder.

h) 4-ary 2-cube med 64 noder.

(Se figur 6.5.)

I tillegg til disse er det også gjort simuleringer av enkle 4x4- og 16x16-svitsjer for å kunne sammenligne med resultatene fra [Bothner 94].



Figur 6.5: Topologiene som er simulert: a) Enkel svitsj med 8 noder. b) 8-svitsjers multi-stage med 32 noder. c) 8+4-svitsjers indirekte multi-stage med 32 noder. d) 8+2-svitsjers indirekte multi-stage med 48 noder. e) 3x3-matrise med 48 noder. f) 4x4-matrise med 80 noder. g) 3-ary 2-cube med 36 noder. h) 4-ary 2-cube med 64 noder. OBS! For hver topologi er det to parallelle nett, ett nett for requester og ett for responser. Alle linker er bidireksjonale. Topologinavnene henspiller på plasseringen av svitsjene.

6.6.1 I programmet

I selve programmet foregår konstruksjonen av nettet slik:

1. Innlesing av topologifil.
2. Lag alle *Switch*-objektene (request- og response-nettet parallelt) med X antall *HIC_interface* i hver *Switch.*

   Lenk sammen (*HIC_interface'*ene i) *Switch*-objektene.
3. Innlesing av labelfil og oppretting av *Node*-objektene.

   Gå igjennom hver *Switch*. Opprett *Node*-objekter (med *HIC_interface*) til alle tomme linker (det vil si de som ikke går til et annet *Switch*-objekt).

- Alle *HIC_interface*-objektene blir bygget opp av andre objekter tilsvarende listene i avsnitt 6.1.3 for henholdsvis noder og svitsjer.

6.6.2 Filsyntaks

Det er påkrevet en spesiell syntaks for topologi- og labelfilene for at simulatoren skal fungere. Det er også viktig at disse filene stemmer overens. Nærmere beskrivelse av filsyntaksen og bruken av simulatoren er gitt i appendiks C.

6.7 Randomfunksjoner

I en simulator er det som regel bruk for tilfeldige (random) tall. I denne simulatoren er det benyttet randomfunksjoner fra GNU's C++-bibliotek. De er hentet fra filene: MLCG.h, NegExp.h, Normal.h og Uniform.h. Nærmere informasjon om disse finnes i [C++ lib 92]. Klassen *MLCG* er selve generatoren og er en implementasjon av en dobbel «Multi Linear Congruential Generator» slik den er beskrevet i [L'Ecuyer 88]. Denne generatoren har en ganske lang periode, og statistiske analyser har gitt god mellom-samplingsuavhengighet [C++ lib 92]. Klassene *NegativeExpntl*, *Normal* og *Uniform* bruker MLCG-generatoren. Funksjonen *time()* fra fila time.h er brukt som frø (seed) til MLCG. *time()* returnerer tiden i sekunder siden 00.00.00 GMT, 1. januar 1970.

Randomfunksjonene er anvendt i to tilfeller. Det ene er for å variere nettverksbelastningen, hvor ofte en request-pakke blir generert (avsnitt 6.4). Det andre er for å trekke mottageradressen for request-pakkene (avsnitt 6.2).

7

Resultater

I forrige kapittel ble selve simulatoren beskrevet, og det ble forklart hvordan ulike statistiske data blir samlet inn og resultater beregnet. Dette kapitlet presenterer resultatene fra simuleringene. Som tidligere nevnt er det simulert 8 forskjellige nettverkstopologier med 8x8-svitsjer (figur 6.5). I tillegg er det gjort simuleringer av enkle 4x4- og 16x16-svitsjer.

Først i kapitlet er det gjort noen teoretiske beregninger av latency og båndbredde, for senere å kunne sammenligne med resultatene fra simuleringene. Videre er det gitt en oppsummering hvor resultatene for de forskjellige nettverkstopologiene blir sammenlignet med hverandre, med en enkel SCI-ring fra [HulBot 93] og med resultatene fra [Bothner 94]. Til slutt er usikkerhetsmomenter ved resultatene diskutert.

7.1 Latency

Som tidligere nevnt, er latency tiden fra starten på en pakke blir sendt til hele pakka er lest av mottagernoden. Den minste mulige latency i et nett får man mellom to noder som henger på samme svitsj, og når pakka ikke må vente på at en link skal bli ledig. I dette kapitlet er det brukt følgende notasjoner ved beregning av latency, verdiene er hentet fra tabell 6.1:

$Nd =$ Node delay, forsinkelsen gjennom et HIC-interface $= 40$ ns
$Ld =$ Link delay, forsinkelsen over en link $= 2$ ns
$Sd =$ Switch delay, forsinkelsen gjennom en svitsj $= 84$ ns
$Pd =$ Packet delay, tiden det tar å lese en hel pakke $= 81 * 12$ ns $= 972$ ns

Minste mulige latency er da gitt ved summen av forsinkelsen i avsender- og mottagernoden ($2Nd$), forsinkelsen over linkene ($2Ld$), forsinkelsen gjennom svitsjen (Sd) og tiden det tar å klokke inn hele pakka (Pd):

$$2Nd + 2Ld + Sd + Pd = 2 * 40 + 2 * 2 + 84 + 972 \text{ ns} = 1128 \text{ ns} \qquad (7.1)$$

I tillegg må man legge til litt forsinkelse for sending av flytkontroll (FCC). Minste simulerte latency er for alle simuleringene 1172 ns, som skulle passe med dette. Tiden det tar å lese inn pakka utgjør en vesentlig del.

Gjennomsnittlig minimumslatency (L_{min}) for en hel nettverkstopologi er regnet ut ved å ta gjennomsnittet av forsinkelsene til hver pakke når en node sender en pakke til alle andre noder i nettet. Teoretisk maksimum latency uten kollisjoner (L_{max}) er gitt ved den lengste veien en pakke må gå mellom to noder.

Eksempel:
Her er det gitt et eksempel på utregning av L_{min} og L_{max} for 8-svitsjers multistage-nettet med 32 noder (figur 6.5 b).

Hvis en node sender en pakke til alle andre noder i nettet, vil den sende til 3 noder over 2 linker og 1 svitsj, til 16 noder over 3 linker og 2 svitsjer og til 12 noder over 4 linker og 3 svitsjer. Tilsammen blir det sendt 31 pakker til 31 noder. Gjennomsnittlig minimuslatency (L_{min}) blir da:

$$L_{min} = \frac{3}{31}(2Nd + 2Ld + Sd + Pd)$$

$$+\frac{16}{31}(2Nd + 3Ld + 2Sd + Pd) + \frac{12}{31}(2Nd + 4Ld + 3Sd + Pd)$$

$$= 2 * 40 + \frac{3(2 * 2 + 84) + 16(3 * 2 + 2 * 84) + 12(4 * 2 + 3 * 84)}{31} + 972 \text{ ns}$$

$$= 1249 \text{ ns}$$

Den lengste veien en pakke må gå, er over 4 linker og 3 svitsjer, slik at maksimum latency uten kollisjoner (L_{max}) blir:

$$L_{max} = 2Nd + 4Ld + 3Sd + Pd$$

$$= 2 * 40 + 4 * 2 + 3 * 84 + 972 \text{ ns} = 1312 \text{ ns}$$

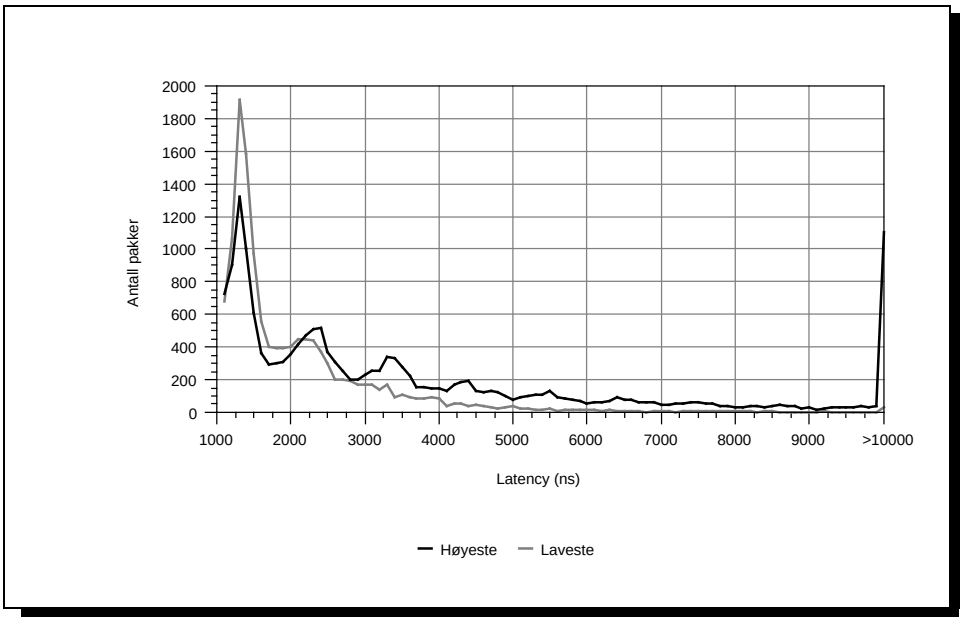
7.1.1 Latency-fordeling

I figur 7.1 er det vist et eksempel på latency-fordelinger av pakker for 4x4-matrisen med 80 noder. Figuren viser fordelingene for simuleringene med høyeste og laveste påtrykk ved 1 utestående request. Vi ser at en vesentlig del av pakkene har forholdsvis lav latency, men ved høyt påtrykk får vi flere pakker med høy latency. Pakker med latency over 10000 ns er summert opp. I resultatene fra simuleringene er gjennomsnittlig latency brukt. For fordelingene fremstilt i figur 7.1, varierer denne fra 2129 ns for laveste påtrykk til 3924 ns for høyeste påtrykk.

7.2 Maksimum teoretisk båndbredde

Den rå båndbredden over en HIC-link (HS) er 1Gbit/s. Siden hver karakter er kodet med 8B/12B-koding, er det kun $\frac{8}{12}$ som er effektive data. Det er også ca 3% av båndbredden som går med til flytkontroll. For hver 32. karakter blir det sendt en FCC. Dette gir $\frac{32}{33}$ effektive data, så tilgjengelig båndbredde (t) for en pakke over en link blir:

$$t = \frac{8}{12} * \frac{32}{33} \text{ Gbits/s} = 80.8 \text{ Mbytes/s} \tag{7.2}$$



Figur 7.1: Latency-fordeling av pakker for simuleringene med høyeste og laveste påtrykk ved 1 utestående request for 4x4-matrisen med 80 noder. Pakker med latency over 10000 ns er summert opp.

I en pakke på 81 karakterer er det kun 64 som er effektive data slik at maksimum effektiv båndbredde over en link (t_{max}) blir:

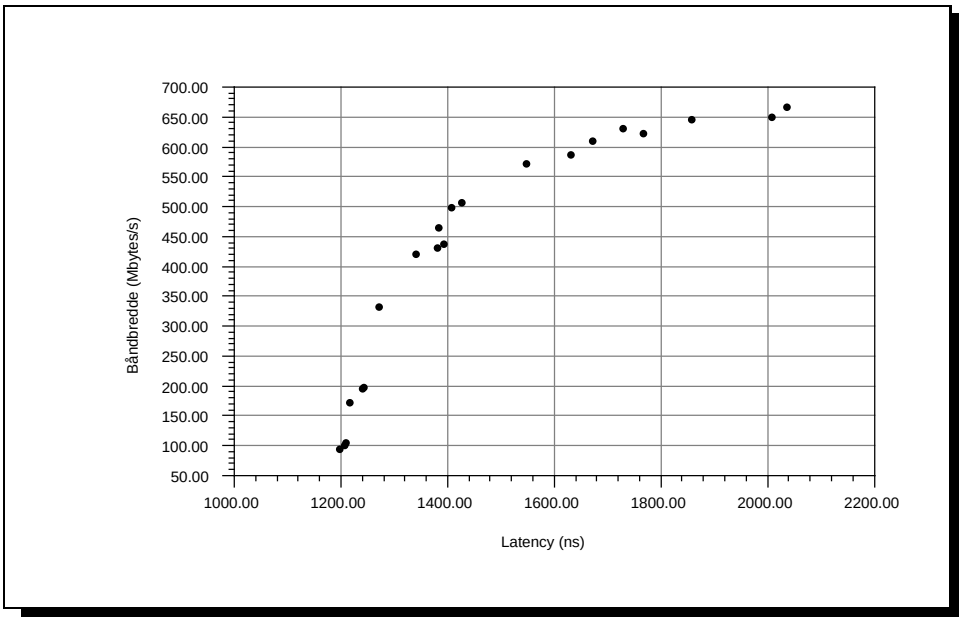
$$t_{max} = \frac{64}{81} * t = \frac{64}{81} * \frac{8}{12} * \frac{32}{33} \text{ Gbits/s} = 63.8 \text{ Mbytes/s} \tag{7.3}$$

I [Inmos 93] er teoretisk båndbredde for et helt system regnet ut ved å summere opp maksimumsbåndbredden for hver input-link til systemet. Med maksimum båndbredde på hver link lik t_{max} fra ligning 7.3, og N antall noder med 2 input-linker fra hver, en for requester og en for responser, blir maksimumsbåndbredden for det totale systemet (T_{max}):

$$T_{max} = 2 * N * t_{max} \tag{7.4}$$

7.3 Simulering av systemer med 8x8-svitsjer

Dette avsnittet presenterer båndbredden (utnyttelsesgraden) som funksjon av latency for alle systemene med 8x8-svitsjer vist i figur 6.5. Alle systemene er simulert med forskjellige påtrykk. Fra det minste påtrykket med 1 utestående request og maksimum forsinkelse mellom response og ny request på 20000 ns, til det største påtrykket med 4 utestående requester og maksimum forsinkelse mellom response og ny request på 200 ns. Resultatet av simuleringene er vist i figurene 7.2–7.9. Punktene i figurene representerer gjennomsnittlig latency og gjennomsnittlig båndbredde for de forskjellige påtrykkene. Aksene i figurene er skalert forskjellig for bedre å kunne se hvordan båndbredden stiger og flater ut. I de følgende avsnittene blir også de teoretiske verdiene for båndbredde og latency beregnet.



Figur 7.2: Latency versus båndbredde for enkel svitsj med 8 noder.

7.3.1 Enkel svitsj med 8 noder

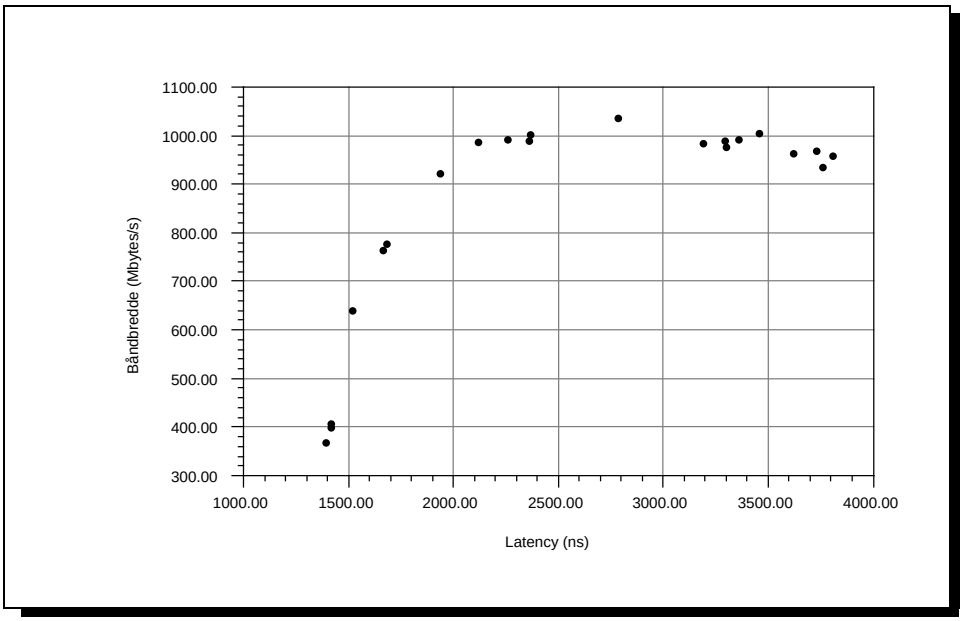
Figur 7.2 viser latency versus båndbredde for et system med 8 SCI-noder knyttet sammen med enkle 8x8-svitsjer, en for request-nettet og en for response-nettet. Vi ser et mønster i at båndbredden først stiger til et visst nivå og så flater ut, systemet går i metning, ved ca 600 Mbytes/s. Vi ser også at latencyen ved liten belastning nærmer seg teoretisk minimumslatency. Ved høy belastning øker latencyen. Hvis vi regner ut den teoretiske maksimumsbåndbredden gitt ligning 7.4, får vi:

$$T_{max} = 2 * 8 * t_{max} = 16 * 63.8 \text{ Mbytes/s} = 1020.8 \text{ Mbytes/s}$$

T_{max} er mye høyere enn resultatet fra simuleringen. Dette kommer av at det oppstår forsinkelser i svitsjen når flere ønsker å sende til samme node. Simulert båndbredde utgjør i underkant av 60% av T_{max}.

Siden nettet her bare består av en svitsj, vil L_{min} og L_{max} være lik minste mulige latency regnet ut i ligning 7.1.

$$L_{min} = L_{max} = 1128 \text{ ns}$$



Figur 7.3: Latency versus båndbredde for 8-svitsjers multi-stage-nett med 32 noder.

7.3.2 Multi-stage med 32 noder

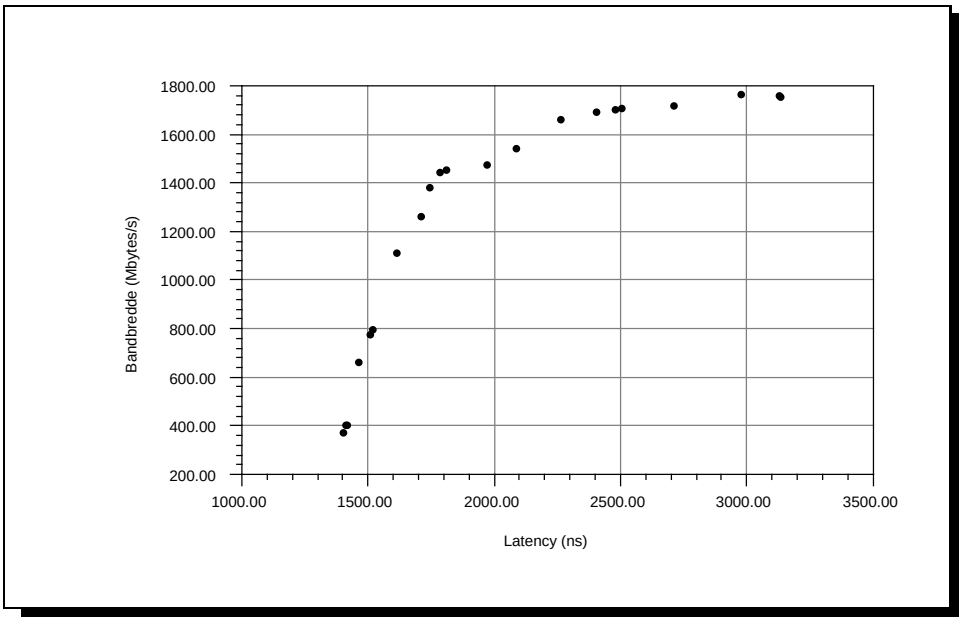
Ved å beregne de teoretiske verdiene som forklart i eksempelet i avsnitt 7.1 og ligning 7.4, får vi følgende for et 8-svitsjers multi-stage-nett med 32 noder:

$$T_{max} = 2 * 32 * t_{max} = 64 * 63.8 \text{ Mbytes/s} = 4083.2 \text{ Mbytes/s}$$

$$L_{min} = 2Nd + \frac{3(2Ld + Sd) + 16(3Ld + 2Sd) + 12(4Ld + 3Sd)}{31} + Pd = 1249 \text{ ns}$$

$$L_{max} = 2Nd + 4Ld + 3Sd + Pd = 1312 \text{ ns}$$

Sammenligner vi med resultatene i figur 7.3, ser vi at den simulerte båndbredden flater ut ved ca 1000 Mbytes/s. Dette er en utnyttelse på 24% i forhold til T_{max}. Latencyen er 1395 ns ved det minste simulerte påtrykket, og i overkant av 2000 ns ved metning. Dette tyder på at det oppstår en del kollisjoner i svitsjene.



Figur 7.4: Latency versus båndbredde for 8+4-svitsjers indirekte multi-stage-nett med 32 noder.

7.3.3 Indirekte multi-stage med 32 noder

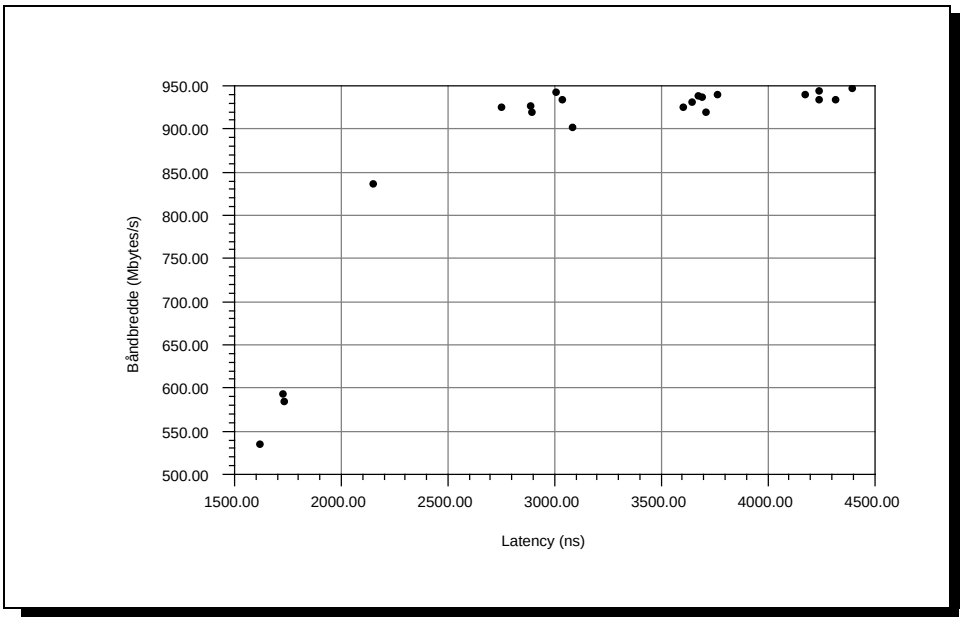
Resultatet fra simuleringen av 8+4-svitsjers indirekte multi-stage-nett med 32 noder er vist i figur 7.4. Vi kan se at det kun er for de aller største påtrykkene at systemet går i metning. Kurven flater her ut ved ca 1600 Mbytes/s. Ved å beregne teoretiske verdier på samme måte som for forrige topologi får vi følgende:

$$T_{max} = 2 * 32 * t_{max} = 64 * 63.8 \text{ Mbytes/s} = 4083.2 \text{ Mbytes/s}$$

$$L_{min} = 2Nd + \frac{3(2Ld + Sd) + 28(4Ld + 3Sd)}{31} + Pd = 1295 \text{ ns}$$

$$L_{max} = 2Nd + 4Ld + 3Sd + Pd = 1312 \text{ ns}$$

Simulert maksimum båndbredde utgjør her 41% av T_{max}. Den minste simulerte latencyen er 1402 ns. Ved metning ser vi latencyen ligger på ca 2250 ns.



Figur 7.5: Latency versus båndbredde for 8+2-svitsjers indirekte multi-stage-nett med 48 noder.

7.3.4 Indirekte multi-stage med 48 noder

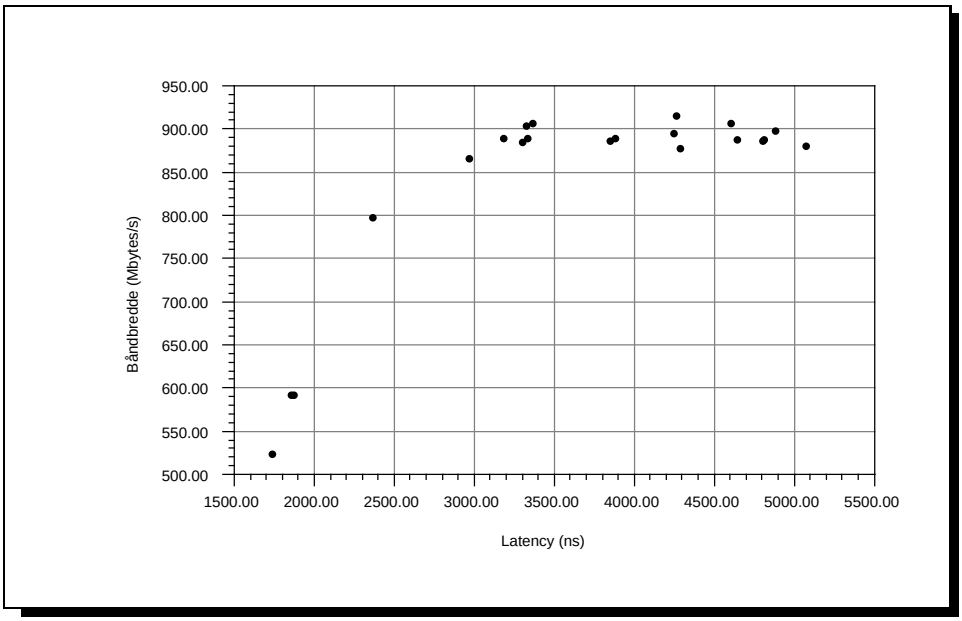
Figur 7.5 viser resultatene for 8+2-svitsjers indirekte multi-stage-nett med 48 noder. Her går nettet raskt i metning. Det er kun kjøringene med de fire laveste påtrykkene som ikke har drevet systemet i metning. Maksimum båndbredde er her 925 Mbytes/s. Teoretiske verdier blir:

$$T_{max} = 2 * 48 * t_{max} = 96 * 63.8 \text{ Mbytes/s} = 6124.8 \text{ Mbytes/s}$$

$$L_{min} = 2Nd + \frac{5(2Ld + Sd) + 24(4Ld + 3Sd) + 18(4Ld + 3Sd)}{47} + Pd = 1290 \text{ ns}$$

$$L_{max} = 2Nd + 4Ld + 3Sd + Pd = 1312 \text{ ns}$$

Utnyttelsen i forhold til T_{max} er her bare 15%. Latencyen er høy og øker veldig når systemet går i metning. Minste simulerte latency er 1618 ns.



Figur 7.6: Latency versus båndbredde for 3x3-matrise med 48 noder.

7.3.5 3x3-matrise med 48 noder

Figur 7.6 viser resultatene for et 3x3-matrise-nett med 48 noder. Her går også systemet relativt raskt i metning, og vi ser kurven flater ut ved 900 Mbytes/s.

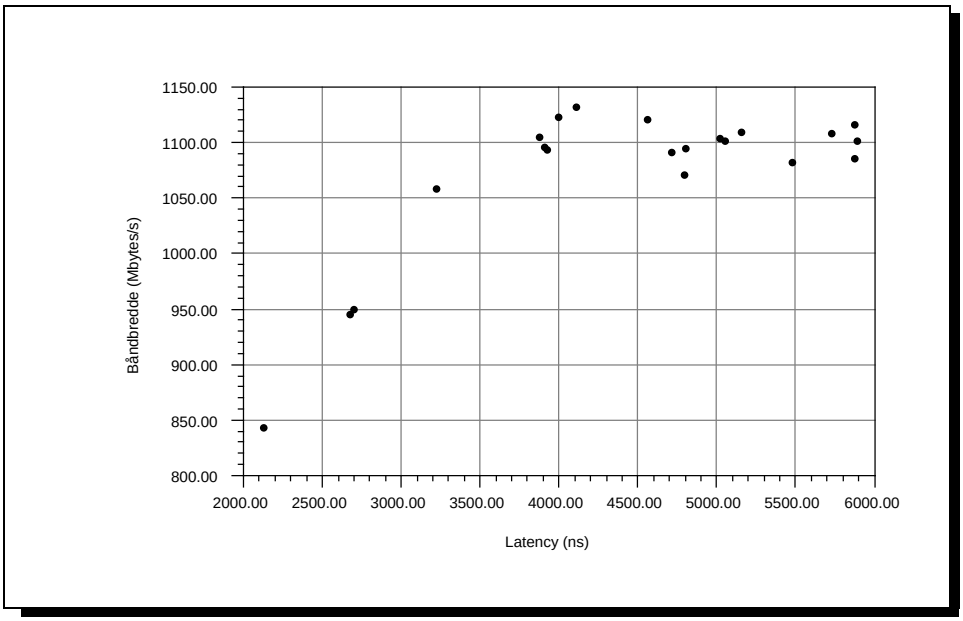
Her blir også teoretiske verdier beregnet som forklart i avsnitt 7.1 og 7.2, men det blir litt mer komplisert. Når man beregner L_{min}, må man ta hensyn til om noden henger på en hjørne-svitsj, side-svitsj eller i midten (se figur 6.5 e). For ikke å fylle opp dette kapitlet med tall, er utregningen skrevet i appendiks D.

$$T_{max} = 2 * 48 * t_{max} = 96 * 63.8 \text{ Mbytes/s} = 6124.8 \text{ Mbytes/s}$$

$$L_{min} = 1301 \text{ ns}$$

$$L_{max} = 2Nd + 6Ld + 5Sd + Pd = 1484 \text{ ns}$$

Simulert båndbredde utgjør også her 15% av T_{max}. Minste simulerte latency er 1739 ns. Ved metning er latencyen over 3000 ns, og den øker raskt.



Figur 7.7: Latency versus båndbredde for 4x4-matrise med 80 noder.

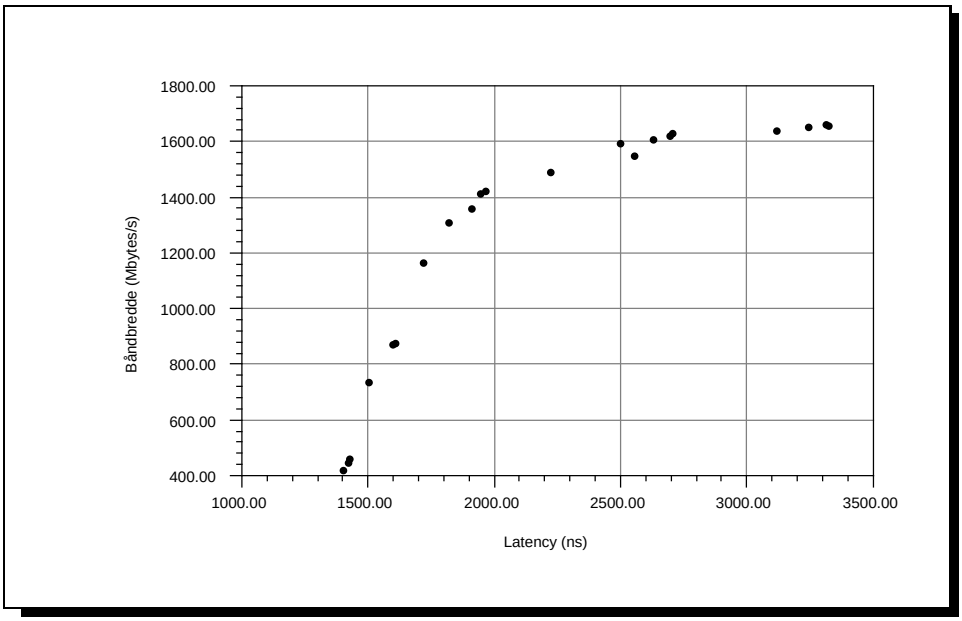
7.3.6 4x4-matrise med 80 noder

Resultatene for et 4x4-matrise-nett med 80 noder er vist i figur 7.7. Her går det også raskt i metning. Båndbredden flater ut ved 1100 Mbytes/s. Dette er en utnyttelse på bare 11% i forhold til T_{max}. Minste latency er 2129 ns. Ved metning er latencyen oppe i hele 4000 ns. Som for 3x3-matrisen er detaljert utregning av L_{min} gitt i appendiks D.

$$T_{max} = 2 * 80 * t_{max} = 160 * 63.8 \text{ Mbytes/s} = 10208.0 \text{ Mbytes/s}$$

$$L_{min} = 1367 \text{ ns}$$

$$L_{max} = 2Nd + 8Ld + 7Sd + Pd = 1656 \text{ ns}$$



Figur 7.8: Latency versus båndbredde for en 3-ary 2-cube med 36 noder.

7.3.7 3-ary 2-cube med 36 noder

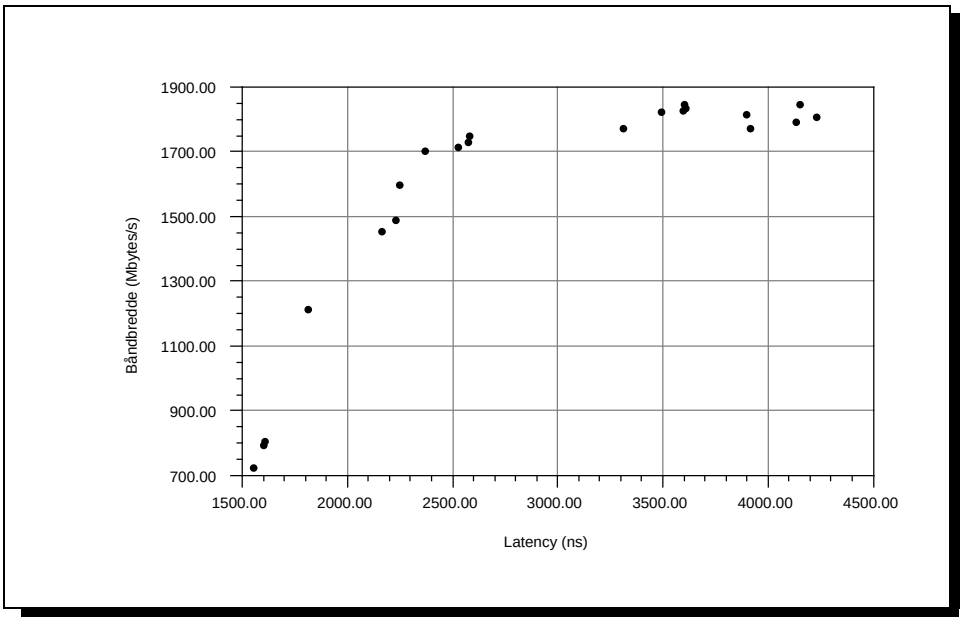
Figur 7.8 viser resultatene for en 3-ary 2-cube med 36 noder. Båndbredden flater ut ved 1500 Mbytes/s. Minste simulerte latency er 1402 ns, og ved metning er latencyen omtrent 2300 ns. Beregning av teoretiske verdier gir:

$$T_{max} = 2 * 36 * t_{max} = 72 * 63.8 \text{ Mbytes/s} = 4593.6 \text{ Mbytes/s}$$

$$L_{min} = 2Nd + \frac{3(2Ld + Sd) + 16(3Ld + 2Sd) + 16(4Ld + 3Sd)}{35} + Pd = 1258 \text{ ns}$$

$$L_{max} = 2Nd + 4Ld + 3Sd + Pd = 1312 \text{ ns}$$

1500 Mbytes/s simulert båndbredde utgjør 32% av T_{max}.



Figur 7.9: Latency versus båndbredde for 4-ary 2-cube med 36 noder.

7.3.8 4-ary 2-cube med 64 noder

Resultatene for en 4-ary 2-cube med 64 noder er vist i figur 7.9. Systemet begynner å gå i metning ved 1700 Mbytes/s. Dette tilsvarer 21% av T_{max}. Minste latency er 1557 ns. Ved metning er latencyen ca 2400 ns. Teoretiske verdier blir her:

$$T_{max} = 2 * 64 * t_{max} = 128 * 63.8 \text{ Mbytes/s} = 8166.4 \text{ Mbytes/s}$$

$$L_{min} = 2Nd + \frac{3(2Ld + Sd) + 16(3Ld + 2Sd) + 24(4Ld + 3Sd)}{63}$$

$$\ldots \frac{+16(5Ld + 4Sd) + 4(6Ld + 5Sd)}{} + Pd = 1315 \text{ ns}$$

$$L_{max} = 2Nd + 6Ld + 5Sd + Pd = 1484 \text{ ns}$$

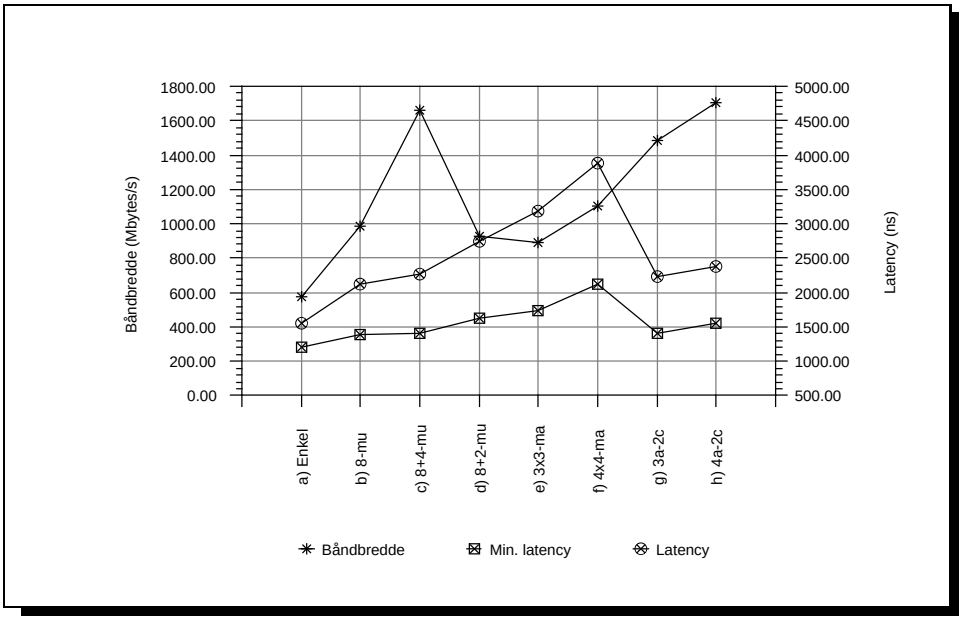
7.4 Gjennomsnittlig trafikk på linkene

Som beskrevet i kapittel 6.5, er det også beregnet gjennomsnittlig trafikk på linkene. Siden det er latency og båndbredde som har vært mest interessant å se på, er ikke disse resultatene brukt til annet enn å sjekke at flytkontrollen (FCC) utgjør ca 3% av båndbredden på hver link. For eksempel ga simuleringen av 4x4-matrisen med størst påtrykk en gjennomsnittlig trafikk på hver link på 62.497 Mchars/s der FCC utgjorde 1.891 Mchars/s, som tilsvarer 3.03%.

7.5 Oppsummering for systemene med 8x8-svitsjer

I følge [Hwang 93] er det fem faktorer som er interessante og kikke på i forbindelse med et nettverks ytelse: Funksjonalitet, latency, båndbredde, hardware-kompleksitet og skalerbarhet. Funksjonalitet og skalerbarhet er ivaretatt av SCI- og HIC-standardene, dette er derfor ikke diskutert i denne oppgaven. Vi skal imidlertid se på de tre andre faktorene, og sammenligne de forskjellige nettverkene som er simulert, ut fra disse. Det er ønskelig med med høy ytelse og lave kostnader. Det vil si lav latency og høy båndbredde kombinert med minst mulig hardware.

En grafisk fremstilling av resultatene fra simuleringene er gitt i figur 7.10. De eksakte tallene er listet opp i tabell 7.1 sammen med de teoretiske verdiene regnet ut i de foregående avsnittene. Verdiene bør ses i sammenheng med de enkelte



Figur 7.10: Latency og båndbredde for de forskjellige systemene med 8x8-svitsjer (vist i figur 6.5). Kurven for båndbredde er knyttet til den venstre aksen, og kurvene for latency er knyttet til den høyre. Verdiene for latency og båndbredde er tatt fra knekkpunktet på grafene i avsnitt 7.3, det vil si der båndbredden flater ut og systemet går i metning. Minste latency er punktet med minst latency fra samme graf.

Topologi	Ant. noder	Ant. sw.	Simulert latency minste ns	Simulert latency ved metn. ns	Teoretisk latency L_{min} ns	Teoretisk latency L_{max} ns	Simulert bb ved metn. MBytes/s	Teoretisk bb T_{max} MBytes/s	Utn. bb %
a) Enkel	8	2	1198	1548	1128	1128	572.4	1020.8	56
b) 8-mu	32	16	1395	2121	1249	1312	986.4	4083.2	24
c) 8+4-mu	32	24	1402	2263	1295	1312	1660.7	4083.2	41
d) 8+2-mu	48	20	1618	2748	1290	1312	925.1	6124.8	15
e) 3x3-ma	48	18	1739	3185	1301	1484	888.9	6124.8	15
f) 4x4-ma	80	32	2129	3875	1367	1656	1104.6	10208.0	11
g) 3a-2c	36	18	1402	2225	1258	1312	1486.9	4087.5	32
h) 4a-2c	64	32	1557	2372	1315	1484	1703.4	8166.4	21

Tabell 7.1: Simulerte resultater og teoretiske verdier for systemene med 8x8-svitsjer vist i figur 6.5. Resultatene fra simuleringene er også plottet i figur 7.10.

plottene i avsnitt 7.3. Vi ser at båndbredde og latency varierer for de forskjellige systemene. Den enkle svitsjen skiller seg litt fra de andre da denne har færre antall noder og linker. Den gir lav båndbredde (572.4 MBytes/s), men utnyttelsen av båndbredden (56%) i forhold til teoretisk båndbredde, T_{max} (1020.8 MBytes/s), er bedre enn for de andre. For multi-stage-nettet (8-mu) får vi en akseptabel latency (2121 ns), men forholdsvis dårlig utnyttelse (24%) av båndbredden i forhold til T_{max} (4083.2 MBytes/s). 8+4-multi-stage-nettet (8+4-mu), 3-ary 2-cube'en (3a-2c) og 4-ary 2-cube'en (4a-2c) gir forholdsvis lav latency (2225–2372 ns) og høy båndbredde (1486.9–1703.4 MBytes/s). 8+4-mu og 3a-2c gir også brukbar utnyttelse (41% og 32%) i forhold til T_{max} (4083.2 og 4087.5 MBytes/s). 8+2-multi-stage-nettet (8+2-mu), 3x3- og 4x4-matrisen (3x3-ma og 4x4-ma) gir forholdsvis høy latency (2748–3875 ns) og lav båndbredde (888.9–1104.6 MBytes/s). Hvis vi ser på figurene 7.5–7.7, ser vi at det bare er de aller laveste påtrykkene som ikke har drevet systemene i metning. Det viser at 8+2-mu, 3x3-ma og 4x4-ma går raskere i metning enn de andre. De har også en dårlig utnyttelse (11–15%) i forhold til teoretisk båndbredde, T_{max} (6124.8–10208.0 MBytes/s).

At utnyttelsen i forhold til T_{max} er best for den enkle svitsjen, kan tyde på at det er vanskeligere å forutsi noe for mer kompliserte nettverkstopologier. De teoretiske verdiene vil kun gi en romslig øvre grense for maksimumsbåndbredden [Horn2 94].

Når det gjelder hardware-kompleksitet, er det matrisene som har flest noder i forhold til svitsjer og linker (figur 6.5), men matrisene gir også dårligst ytelse. Ved å gi matrisene en tilbakekobling slik at man får k-ary 2-cubes som i 3a-2c og 4a-2c, er det oppnådd bedre ytelse, men man får færre noder i forhold til antall svitsjer og linker. Av multi-stage-nettene er det 8+4-mu som gir best ytelse. 8+4-mu har større hardware-kompleksitet, det vil si flere svitsjer og linker i forhold til antall noder. Derfor er det større mulighet til en bedre fordeling av trafikken. 8-mu og 8+2-mu har færre linker og svitsjer, og har derfor begrenset ytelse som en følge av topologi og rutingalgoritme.

Alle systemene gir forholdsvis lav latency (1198–2129 ns) ved lavt påtrykk, her utgjør pakkelengden en vesentlig del (972 ns). Når belastningen øker og systemene går i metning stiger latencyen drastisk. Alle systemene har høyere simulerte verdier for latency enn både teoretisk gjennomsnitt (L_{min}) og teoretisk maksimum latency

(L_{max}) (tabell 7.1). Det viser at det oppstår kollisjoner i nettet. Vi ser også at det er en sammenheng mellom høy latency, dårlig utnyttelse av båndbredden og liten hardware-kompleksitet. Flaskehalsene er stort sett mellom svitsjene. Spesielt vil dette påvirke topologier der antall linker er begrenset.

8+2-mu og 3a-2c kommer totalt sett best ut. 3a-2c har flere noder og færre svitsjer enn 8+2-mu. Noe som gir mindre hardware-kompleksitet (figur 6.5).

Det er i denne oppgaven brukt en jevn distribuering av pakker i nettet. Det er mulig man får en bedre utnyttelse hvis man regner med større sannsynlighet for at en pakke blir sendt til en node som ligger nærmere, for eksempel på samme svitsj.

7.6 Sammenligning med [HulBot 93]

For å vise påliteligheten av resultatene i denne oppgaven, er det naturlig å sammenligne med hva andre har gjort på området. I utgangspunktet var det meningen å sammenligne med SCI-systemene undersøkt i [HulBot 93], men dette er vanskelig siden det der er snakk om rene SCI-systemer som gir bedre ytelse. SCI kjører 16 bit parallelt mot HIC som kjører kun 1 bit serielt. I [HulBot 93] er det også simulert større topologier med mange noder.

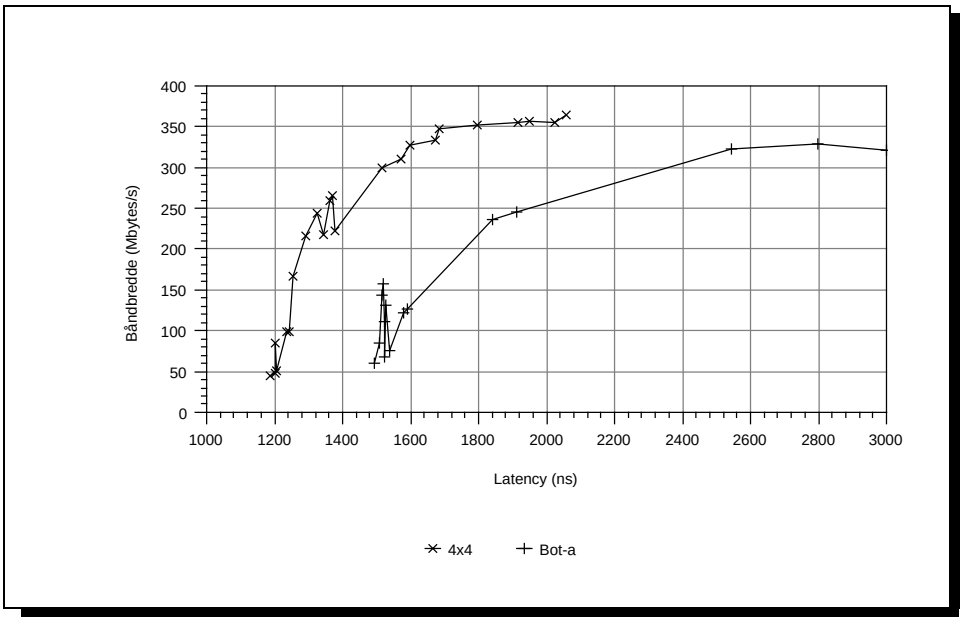
Hvis vi sammenligner den enkle 8x8-svitsjen med 8 noder, med en enkel SCI-ring med 8 noder simulert i [HulBot 93], kommer SCI-ringen vesentlig bedre ut både når det gjelder båndbredde og latency. SCI-ringen har en maksimumsbåndbredde på 1300 Mbytes/s mot ca 600 Mbytes/s for HIC-svitsjen. Minste latency er 110 ns mot 1198 ns, og ved metning gir SCI-ringen en latency på 540 ns mot 1548 ns for HIC-svitsjen.

Resultatene her sammenlignet med resultatene fra [HulBot 93] virker derfor sannsynlige. Et SCI-system med HIC som transportmedium kan ikke gi samme ytelsen som et rent SCI-system. Båndbredden i en enkel SCI-ring er derimot konstant [HulBot 93]. Derfor kan et HIC-system kunne gi høyere båndbredde hvis antall noder i en SCI-ring blir mange. HIC-systemet vil likevel være vesentlig dårligere når det gjelder latency. En fordel med HIC-systemet er at hvis en node eller link faller ut, vil resten av systemet fortsatt kunne være operativt. Hvis en node eller link i en SCI-ring faller ut, vil hele ringen gå ned.

7.7 Enkle svitsjer sammenlignet med [Bothner 94]

Simulatoren brukt i [HulBot 93] er senere blitt modifisert. Først for å simulere en enkel HIC-forbindelse mellom to SCI-ringer [Krist 93], og så for å kunne simulere en HIC-svitsj som knytter flere SCI-noder eller ringer sammen [Bothner 94]. Det er derfor mest interessant å sammenligne med resultatene i [Bothner 94]. Det er verdt og merke seg at [Bothner 94] simulerer kun SCI-delen med innlagte forsinkelser for HIC, mens simuleringene i denne oppgaven er gjort for HIC-delen med forsinkelser for det som skjer i SCI-delen. I [Bothner 94] er det også tatt utgangspunkt i et system med en SCI/HIC-bro[1]. Derfor er det benyttet litt andre forsinkelsesparametere. Minste mulige teoretiske latency er i [Bothner 94] regnet ut til å være 1436 ns. Det

[1] En slik bro kan for eksempel være SCI/HIC-interfacet beskrevet i [Tørudb 94].



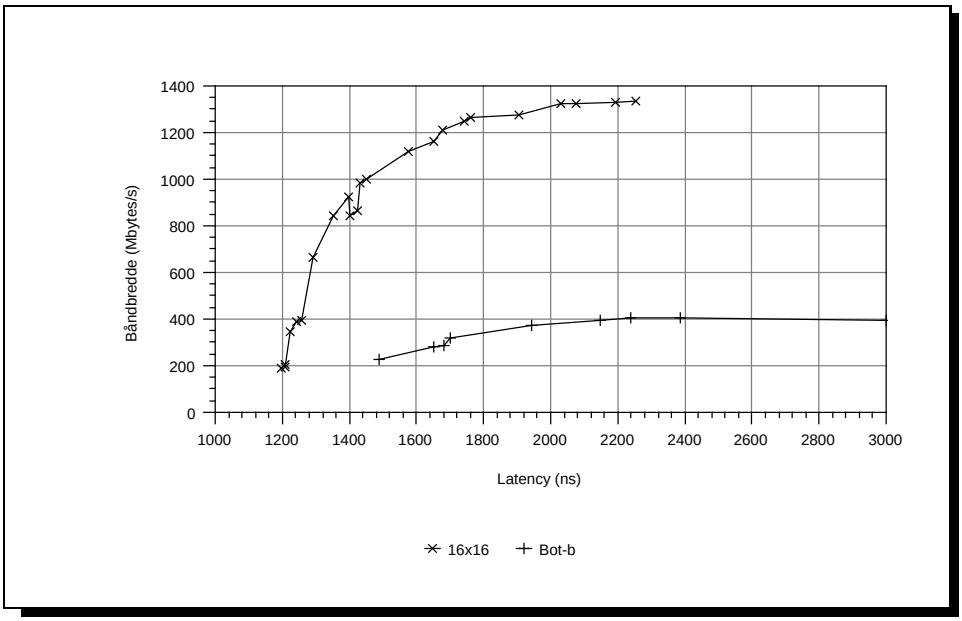
Figur 7.11: Simulering av 4x4-svitsj sammenlignet med Bot-a fra [Bothner 94].

er 308 ns mer enn minste latency på 1128 ns regnet ut i ligning 7.1. Ved utregningen av teoretisk båndbredde, er det i [Bothner 94] gjort et par feil[2].

I [Bothner 94] er det simulert en 8x8-svitsj som tar seg av både request-delen og response-delen. Dette gir kun mulighet til å koble 4 SCI-ringer eller noder på en svitsj. Det er gjort simuleringer av to systemer, et med 4 noder «direkte» på svitsjen (Bot-a) og et med 4 ringer kyttet til svitsjen med 4 noder på hver ring (Bot-b). For å ha noe å sammenligne resultatene fra [Bothner 94] med, er det derfor her kjørt noen ekstra simuleringer av en enkel 4x4-svitsj med 4 noder og en enkel 16x16-svitsj med 16 noder. Resultatene er plottet sammen med resultatene fra [Bothner 94] i figur 7.11 og 7.12.

Som nevnt har [Bothner 94] brukt andre forsinkelsesparametere, noe som gir utslag i lengere latency for Bot-a og Bot-b. Vi ser at differansen er ca 300 ns ved minste påtrykk, noe som er i samsvar med de teoretiske beregningene. Ellers ser vi at systemene med 4 noder i figur 7.11 gir tilnærmet samme resultater for båndbredden. Kurvene for både 4x4-svitsjen og Bot-a flater ut i overkant av 300 Mbytes/s. Systemene med 16 noder i figur 7.12 er det derimot stor forskjell på. 16x16-svitsjen gir en båndbredde på hele 1300 Mbytes/s mot Bot-b som flater ut ved 400 Mbytes/s. Dette tyder på at så lenge pakkene er jevnt distribuert i systemet, er det mer effektivt å knytte nodene direkte sammen med en HIC-svitsj enn å knytte SCI-ringer sammen. Man må kunne forvente en høyere ytelse for Bot-b hvis man øker sannsynligheten for at nodene sender til en node på samme ring. Når det gjelder latency, viser resultatene at man vinner en del ved å hekte HIC-interfacet direkte på SCI-

[2] I avsnitt 6.2 side 25 er det brukt gale verdier for hvor mye av båndbredden som er flytkontroll og hvor mye som er effektive data. [Bothner 94] skriver at flytkontrollen utgjør ca $\frac{1}{80}$ av båndbredden. Riktig verdi er $\frac{1}{33}$. Effektive data i en SCI-pakke utgjør riktignok $\frac{64}{80}$, men det blir da $\frac{64}{82}$ for en HIC-pakke med HIC-hode og EOP slik det er simulert i [Bothner 94].



Figur 7.12: Simulering av 16x16-svitsj sammenlignet med Bot-b fra [Bothner 94].

nodene, i motsetning til bruken av en SCI/HIC-bro som i [Bothner 94].

7.8 Usikkerheter ved resultatene

Det er flere faktorer som innvirker på de statistiske dataene og kan gi usikkerheter ved resultatene. I en simulator blir det gjort forenklinger [Shannon 75]. Man er nødt til å filtrere ut det viktigste. I denne simulatoren er det for eksempel ikke tatt hensyn til fysisk støy eller forskyvning av signalene på linkene.

I en multiprosessorarkitektur med flere prosessorer som arbeider uavhengig av hverandre, kan det være noen som arbeider samtidig. Dette er vanskelig å simulere i et program som blir utført sekvensielt. En måte å løse dette på, og som er brukt i denne oppgaven, er intervallorientert metode. Man øker klokka med et gitt intervall. For hver gang klokka øker, går man gjennom alle komponentene i nettverket og sjekker om noe skal bli utført. Det blir utført hvis så er tilfelle. Dette gir en form for prioritet ved at komponentene blir eksekvert i samme rekkefølge for hver klokkesykel, som nevnt i kapittel 6.4.

En annen usikkerhetsfaktor er simuleringstiden. Alle simuleringene er kjørt med 500000 klokkesykler (1ms), og som beskrevet i kapittel 6.4 skulle dette være godt nok. Likevel er påliteligheten av alle statistiske beregninger avhenging av hvor stort tallmateriale man har. Ved å simulere lenger, vil man få flere pakker gjennom nettverket og dermed større sikkerhet ved resultatene. Det som begrenser dette, er at det tar tid å kjøre en simulering. En simulering kan ta opptil flere timer ved høyt påtrykk og mange noder. Det er også avhengig av hvor stor belastning det er på maskinen simuleringen blir kjørt på.

Hvor ofte en request-pakke blir sendt blir trukket tilfeldig (random). Det betyr

at det er randomfunksjonen (kapittel 6.7) som styrer trafikken i nettverket. Adresseringen av request-pakkene er også trukket med randomfunksjonen. Resultatene er med andre ord avhengige av kvaliteten på randomfunksjonen [Press 88].

Som tidligere nevnt så er det kun HIC-delen av nettverket som i realiteten blir simulert. SCI-delen blir simulert ved å legge til verdier for forsinkelsen gjennom denne. R-FIFO i noden blir også tømt umiddelbart. I et reelt tilfelle kunne man tenke seg at SCI-noden kanskje var opptatt med andre ting, slik at det ville ta litt tid før en pakke ble lest inn. En slik belastningsvariabel er heller ikke brukt i [HulBot 93] eller [Bothner 94].

8

Konklusjon

I oppgaven er det sett på hvordan man kan bruke HIC som et transportmedium for SCI. For å finne ut hvordan ytelsen i slike systemer vil bli, er det laget en simulator. Det er gjort simuleringer av forskjellige nettverk, og det er vurdert forskjellige faktorer rundt bruken av HIC, som pakkeformater, adressering og problemer med deadlock. I simulatoren er det benyttet parametere basert på arbeidet i ESPRIT-prosjektet OMI/HIC.

De viktigste valgene som er gjort, har vært i forbindelse med representasjonen av nodene, svitsjene, og de andre komponentene i simulatoren, implementasjonen av rutingalgoritmene, utviklingen av svitsjen og håndteringen av flytkontrollen.

Resultatene her og i [Krist 93] og [Bothner 94] indikerer at HIC vil være brukbart som transportmedium for SCI. For å oppnå en brukbar ytelse, er det viktig å se på hvordan systemene blir implementert. Forskjellige nettverkstopologier gir forskjellig ytelse. Felles for alle systemene er at de gir lav latency når belastningen er liten, mens latencyen øker når belastningen øker og systemet går i metning.

Som forklart i kapittel 4, må man ta hensyn til muligheten for deadlock. Dette er blant annet gjort ved å bruke doble nett, ett for requester og ett for responser. Implementasjonene er også deadlock-frie med hensyn på topologi og rutingalgoritme, og simuleringene indikerer at systemene ikke bryter sammen ved høy belastning.

Resultatene fra simuleringene stemmer overens med tidligere arbeider [HulBot 93, Bothner 94]. Det har kun vært mulig å sammenligne med enkelte topologier. SINTEF er i ferd med å utvikle en ny simulator [Horn1 94], slik at det vil bli bruk for resultatene fra oppgaven når den simulatoren er ferdig.

Et videre arbeid kan blant annet bestå i å se på lokalitet og hva man oppnår ved å øke sannsynligheten for at en pakke blir sendt til en node som ligger nærmere. Man kan simulere andre nettverkstopologier, og man kan ta for seg andre rutingalgoritmer. Det kan være interessant å se på andre svitsjtyper og -størrelser, og man kan justere parametere når flere data foreligger fra arbeidet med RCUBE.

Referanser

[Bothner 94] John Bothner and Ernst Kristiansen: *«Simulator report for SCI systems with HIC-router»* ISBN 82-595-8732-7, SINTEF Instrumentation, September 1994.

[Bugge 90] Håkon Bugge, Ernst Kristiansen og Bjørn Bakka: *«Trace-Driven Simulations for a Two-level Cache Design in Open Bus Systems»* Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, Washington, May 28-31, 1990.

[BULLIT 94] BULL S.A.: *«BULLIT»* Data Sheet v1.0, © Bull Serial Link Technology, May 1994.

[C++ lib 92] Doug Lea: *«User's Guide to the GNU C++ Library»* Free Software Foundation, Inc., version 2.0, 29. April 1992.

[Dally 87] William J. Dally and Charles L. Seitz: *«Deadlock-Free Routing in Multiprocessor Interconnection Networks»* IEEE Transactions on Computers, vol. C-36, no. 5, May 1987.

[Dolphin 91] Dolphin: *«The Scalable Coherent Interface»* Technology Overview, © Dolphin Server Technology, 1991.

[Dolphin 92] Dolphin: *«DST501A GaAs NodeChip™»* Advance Information Data Sheet, DST501A-A1, © Dolphin SCI Technology, 1992.

[GlassNi 92] Christopher J. Glass and Lionel M. Ni: *«The Turn Model for Adaptive Routing»* Proc. 19th International Symposium on Computer Architecture, IEEE CS Press, California, order no. 2940, 1992.

[Horn1 94] Geir Horn, John Bothner, Svein Linge, Ernst Kristiansen and Øystein Gran Larsen: *«Simulating complex SCI topologies»* SCI meeting, Oslo, 21. September 1994.

[Horn2 94] Geir Horn, Svein Linge, Ernst Kristiansen and Øystein Gran Larsen: *«Performance Simulations of Networks with Point-to-Point Links»* presented at International Data Acquisition Conference on Event Building and Data Readout, Fermilab National Accelerator Laboratory, Batavia, Illinois, USA, 26-28 October 1994.

[HulBot 93] John Bothner og Trond Hulaas: *«Topologies for SCI-based systems with up to a few hundred nodes»* Hovedoppgave ved Institutt for Informatikk, Universitetet i Oslo, 9. februar 1993.

[Hwang 93] Kai Hwang: *«Advanced Computer Architecture: Parallelism, Scalability, Programmability»* ISBN 0-07-031622-8, © McGraw-Hill, Inc., 1993.

[IEEE-HIC 95] IEEE: *«Standard for Heterogeneous InterConnect (HIC)»* IEEE Draft Std P1355, version D1.3, © IEEE New York, 12. January 1995. Tilgjengelig via anonym ftp fra ftp.inmos.co.uk på inmos/projects/ieee/hic/DraftD1.3.ps

[IEEE-SCI 92] IEEE: *«The Scalable Coherent Interface»* IEEE Std 1596, © IEEE New York, 1992.

[Inmos 93] INMOS: *«Network, Routers and Transputers: Function, Performance and Applications»* ISBN 90-5199-129-0, © INMOS Limited, 1993. Tilgjengelig via anonym ftp fra ftp.inmos.co.uk på inmos/info/comms/book/*

[James 94] Dave James: *«Deadlock-free message-fabric routing»* (P1596.1 SCI-switches working-group topic), 25. September 1994.

[KimDas 94] Jong Kim and Chita R. Das: *«Hypercube Communication Delay with Wormhole Routing»* IEEE Transaction on Computers, vol.43, no. 7, 1994.

[Krist 93] Ernst Kristiansen: *«System behavior for SCI and HIC in various topologies»* OMI/HIC ESPRIT project 7252 task 5.1, OMI/HIC project confidential, 21. April 1993.

[L'Ecuyer 88] Pierre L'Ecuyer: *«Efficient and Portable Combined Random Number Generators»* Communications of the ACM, vol. 31, no. 6, 1988.

[LeeuTan 87] J. van Leeuwen and R. B. Tan: *«Interval routing»* The Computer Journal, vol. 30, no. 4, pp. 298-307, 1987.

[NiMcKi 93] Lionel M. Ni and Philip K. McKinley: *«A survey of wormhole routing techniques in direct networks»* Computer, pp. 62-76, February 1993.

[Press 88] William H. Press, Brian P. Flannery, Saul A. Teukolsky and William T. Vetterling: *«Numerical Recipes in C: The Art of Scientific Computing»* ISBN 0-521-35465-X, © Cambridge University Press, 1988.

[RCUBE 94] OMI/HIC: *«RCUBE preliminary data»* Confidential OMI/HIC, Version 1.1, November 1994.

[Shannon 75] R. E. Shannon: *«System Simulation, the art and sience»* ISBN 0-13-8818398, © Prentice-Hall, Inc., 1975.

[ST C104 94] SGS-THOMSON: *«ST C104 asynchronous packet switch»* Preliminary Data, © SGS-THOMSON Microelectronics, June 1994. Tilgjengelig via anonym ftp fra ftp.inmos.co.uk på inmos/info/comms/datasheets/C104-*.ps.Z

[Strous 91] Bjarne Stroustrup: *«The C++ Programming Language 2nd Edition»* ISBN 0-201-53992-6, © Addison-Wesley Publishing Company, Inc., 1991.

[Tanenb 89] Andrew S. Tanenbaum: *«Computer Networks Second Edition»* ISBN 0-13-166836-6, © Prentice-Hall, Inc., 1989.

[Tørudb 94] Ola Tørudbakken: *«The SCI/HIC Interface»* Hovedoppgave ved Institutt for Informatikk, Universitetet i Oslo, 15. november 1994.

[Valiant 82] L. G. Valiant: *«A Sceme For Fast Parallel Comunication»* SIAM Journal on Computing, vol. 11, no. 2, May 1982.

Appendiks A

Ordliste

Her følger en liten ordliste med ord og uttrykk som er brukt i oppgaven. De fleste av dem er låneord eller navn på fenomener tatt fra engelsk.

Bypass Forbipasserende. Brukt i forbindelse med bypass-fifo i en SCI-node. Se kapittel 2.2.

Cache Liten, lokal høyhastighetshukommelse i en prosessor. Blir brukt til midlertidig lagring av data lokalt, for å få lavere tidsforbruk ved hukommelsesaksessering.

Character Her brukt om HIC-character/karakter. Se kapittel 3.2.3, og 3.3.1 om 8B/12B-koding.

Coherens Koherens, sammenheng, forbindelse.

DC-balanse Et DC-balansert kodemønster har en konstant DC-komponent uavhengig av datamønsteret. Dette gir fordeler ved høyhastighetstransmisjoner, for eksempel over fiberoptiske linker [IEEE-HIC 95]. DC = likstrøm.

Deadlock Deadlock er en situasjon der et nettverk er låst. Det vil si at et sett pakker er blokkert i nettet og vil aldri nå frem til mottagernoden.

Delay Forsinkelse.

EOP «End-of-packet». Et merke som angir slutten på en pakke.

Echo eller echo-pakke. SCI transaksjon. Se kapittel 2.2.1.

FCC «Flow control link character». HIC's flytkontroll. Se kapittel 3.2.3.

Fifo «First-in-first-out». En ordnet kø, hvor første inn er første ut. Blir brukt som et buffer for å lagre henholdsvis HIC-karakterer eller SCI-symboler.

Hardware Fysiske komponenter.

Header Hodet på en HIC-pakke. Inneholder mottagernodens adresse.

Idle «Ikke i bruk», ledig.

Input eller input-link. Brukt om innkommende link i node eller svitsj. Også brukt i forbindelse med input-fifo (se fifo).

Interface Overgang mellom to systemer. I dette tilfellet SCI og HIC.

Intervall-ruting Rutingalgoritme beskrevet i kapittel 5.2. Se også ruting.

Label Verdien på en utgang (output-link) i en svitsj i forbindelse med intervall-ruting.

Latency Tiden en pakke bruker gjennom nettet.

Link Fysisk punkt-til-punkt forbindelse mellom noder og/eller svitsjer.

Memory Hukommelse.

Output eller output-link. Brukt om utgående link i node eller svitsj. Også brukt i forbindelse med output-fifo (se fifo).

Payload «Innmaten»/data-feltet i en HIC-pakke. Se kapittel 3.2.4.

Random Tilfeldig. Her brukt i forbindelse med randomfunksjoner/-generatorer. som produserer (mer eller mindre) tilfeldige tall. Se kapittel 6.7.

Random-ruting Rutingalgoritme beskrevet i kapittel 5.3.2. Se også ruting.

Receiver Mottager.

Request eller request-pakke. SCI transaksjon. Se kapittel 2.2.1.

Response eller response-pakke. SCI transaksjon. Se kapittel 2.2.1.

Retry «Nytt forsøk».

Ruting Prosessen å overføre en pakke fra en avsender til mottager. En rutingalgoritme angir reglene for hvor en pakke blir sendt videre fra en node/svitsj.

Stack Stabel. Flere lag oppå hverandre.

Transmitter Avsender.

West-first-ruting Rutingalgoritme beskrevet i kapittel 5.3.1. Se også ruting.

Whormhole-ruting Rutingalgoritme beskrevet i kapittel 5.1. Se også ruting.

Appendiks B

C++

Til implementasjonen av min simulator har jeg valgt programmeringsspråket C++. Det er flere gode grunner til det.

- Det er objekt-orientert. Noe som egner seg godt når man skal beskrive en simulator med flere noder, svitsjer og linker.
- Det gir mulighet for en viss form for parallellitet. Flere noder kan eksistere og utføre ting uavhengig av hverandre.
- Det gir en akseptabel hastighet. Både for kompilering og eksekvering.
- Det er portabelt. Gir muligheter til å flytte mellom forskjellige platform-er/arkitekturer.
- Jeg kunne dra nytte av noe av arbeidet som var gjort i [HulBot 93].
- C++ er blitt veldig utbredt.

Som kompilator har jeg brukt GNU's C++ kompilator, g++. Denne er tilgjengelig for de fleste arkitekturer. Mer info om C++ finnes blant annet i [Strous 91].

Appendiks C

Bruk av simulatoren

Selve programmet som utgjør simulatoren, er kalt HICsim og blir kjørt med kommandoen HICsim <opsjoner>. Her følger en kort beskrivelse av HICsim's opsjoner, hvilke konstanter som er satt og hvordan syntaksen skal være på topologi- og labelfilene.

C.1 Opsjoner

-top fil	Angir topologifila (default = simple.top) Se beskrivelse av filsyntaks i avsnitt C.3.
-l fil	Angir labelfila (default = simple.label) Labelfila må stemme overens med topologifila, se beskrivelse av filsyntaks i avsnitt C.3.
-o fil	Resultatfila (default = resultat.out)
-t n	Lengden på simuleringen i antall sykler, setter variabelen *SIMULATIONTIME* (default = 2000).
-p n	Angir hvor mange utestående requester en node kan ha, setter variabelen *MAX_TRANSACTIONS* (default = 1).
-d c	Velger fordelingsfunksjon som trekker når neste request blir sendt. U = uniform (default), E = neg.exp., N = normal.
-**rd** n	Setter maksimum forinkelse fra response er mottat til ny request blir generert, kun for uniform-fordelingen (default = 200 sykler).
-u	Gir utskrift av hva som skjer hver sykel. Brukt til debugging.
-h	Gir en kort hjelp som denne.

Eksempel:

HICsim -top simple.top -l simple.label -t 500000 -rd 100 -p 4 -o simple-100-4.out

Her blir det simulert en enkel svitsj beskrevet i filene simple.top og simple.label. Simuleringstiden er satt til 500000 sykler, som tilsvarer 1ms. Fordelingsfunksjonen er satt som default til uniform-fordelingen, med maksimumsforsinkelse på 100 sykler. Antall utestående requester er satt til 4. Resultatene fra simuleringen blir skrevet ut på fila simple-100-4.out.

C.2 Konstanter

I tillegg til opsjonene er det satt noen konstanter i kildekoden. Disse konstantene er satt i fila defs.H:

R_FIFO_SIZE	og
T_FIFO_SIZE	setter lengden på henholdsvis *R_FIFO* og *T_FIFO*.
CPU_PROS_TIME	tiden en node bruker på å lage en response etter å ha mottat en request.
LINK_DELAY	forsinkelse over linken.
SWITCH_DELAY	total forsinkelse gjennom svitsjen.
SCI_BULLIT_DELAY	og
BULLIT_DELAY	er slått sammen til forsinkelsen gjennom HIC-interfacet.
CHAR_DELAY	tiden det tar å lese en HIC-karakter.
FCC_UPDATE_DELAY	tid fra en FCC kommer inn til *FCC_ct* er oppdatert.
FCC_GEN_DELAY	tid fra *Read_ct=0* til FCC er generert.

Pakkelengden, pakkeformatet og tiden det tar å klokke inn en pakke, er hardkodet.

C.3 Filsyntaks

For at programmet skal fungere, er det påkrevet en spesiell syntaks for topologi- og labelfilene. Det er viktig at disse filene stemmer overens. Syntaksen er best forklart ved hjelp av et eksempel. Filene multi.top og multi.label inneholder henholdsvis topologibeskrivelsen og lablene for et 8-svitsjers multi-stage nett (figur 6.5 b). Kommentarene er ikke en del av fila.

multi.top:

```
8 8          <-  Topologien består av 8 stk. 8x8-svitsjer (antall først).
0 4          <-  Svitsj nr 0 får 4 «nye» linker
7 5 3 1      <-  til svitsj nr 7, 5, 3 og 1.
1 3          <-  Svitsj nr 1 får 3 «nye» linker
6 4 2        <-  til svitsj nr 6, 4 og 2.
2 3              ...osv.
7 5 3
3 2              Viktig at høyeste svitsj nr av de «nye»
6 4              linkene kommer først.
4 2
7 5
5 1
6
6 1
7
```

multi.label:

```
8  8                          <-  8 stk. 8x8-svitsjer som i multi.top.
0  1  2  3 11 19 27 31        <-  Lablene for link 0–7 i svitsj nr 0.
3  4  5  6  7 15 23 31        <-  Lablene for link 0–7 i svitsj nr 1.
7  8  9 10 11 19 27 31            ...osv.
3 11 12 13 14 15 23 31
7 15 16 17 18 19 27 31
3 11 19 20 21 22 23 31
7 15 23 24 25 26 27 31
3 11 19 27 28 29 30 31        <-  Lablene for link 0–7 i svitsj nr 7.
```

Appendiks D

Utregning for matrisene

D.1 3x3-matrisen

Når man skal regne ut L_{min} for matrisene, må man ta hensyn til plasseringen av nodene. For 3x3-matrisen er det 24 noder på hjørne-svitsjene, 20 noder på side-svisjene og 4 noder på svitsjen i midten, tilsammen 48 noder. Gjennomsnittlig minimumslatency (L_{min}) for hele nettet blir da:

$$L_{min} = 2Nd + Pd$$

$$+\frac{5(2Ld+Sd)+10(3Ld+2Sd)+16(4Ld+3Sd)+10(5Ld+4Sd)}{47}$$

$$\cdots\frac{+6(6Ld+5Sd)}{}*\frac{24}{48}$$

$$+\frac{4(2Ld+Sd)+16(3Ld+2Sd)+15(4Ld+3Sd)+12(5Ld+4Sd)}{47}*\frac{20}{48}$$

$$+\frac{3(2Ld+Sd)+20(3Ld+2Sd)+24(4Ld+3Sd)}{47}*\frac{4}{48}$$

$$= 2*40+972+\frac{24*12392+20*11188+4*9984}{47*48}\text{ ns} = 1301\text{ ns}$$

D.2 4x4-matrisen

For 4x4-matrisen er det 24 noder på hjørne-svitsjene, 40 noder på side-svisjene og 16 noder på svitsjene i midten, tilsammen 80 noder. Gjennomsnittlig minimumslatency (L_{min}) for hele nettet blir da:

$$
\begin{aligned}
L_{min} &= 2Nd + Pd \\
&+ \frac{5(2Ld + Sd) + 10(3Ld + 2Sd) + 14(4Ld + 3Sd) + 20(5Ld + 4Sd)}{79} \\
&\quad \cdots \frac{+14(6Ld + 5Sd) + 10(7Ld + 6Sd) + 6(8Ld + 7Sd)}{} * \frac{24}{80} \\
&+ \frac{4(2Ld + Sd) + 15(3Ld + 2Sd) + 19(4Ld + 3Sd) + 19(5Ld + 4Sd)}{79} \\
&\quad \cdots \frac{+16(6Ld + 5Sd) + 6(7Ld + 6Sd)}{} * \frac{40}{80} \\
&+ \frac{3(2Ld + Sd) + 18(3Ld + 2Sd) + 30(4Ld + 3Sd) + 22(5Ld + 4Sd)}{79} \\
&\quad \cdots \frac{+6(6Ld + 5Sd)}{} * \frac{16}{80} \\
&= 2 * 40 + 972 + \frac{24 * 27592 + 40 * 24496 + 16 * 21664}{79 * 80} \text{ ns} \\
&= 1367 \text{ ns}
\end{aligned}
$$